OPEN ACCESS

KING'S College LONDON

King's Research Portal

*DOI:*
10.1109/TDSC.2019.2909902

*Document Version*
Peer reviewed version

Link to publication record in King's Research Portal

*Citation for published version (APA):*
Bai, C., Han, Q., Mezzour, G., Pierazzi, F., & Subrahmanian, V. S. (2019). DBank: Predictive Behavioral Analysis of Recent Android Banking Trojans. *IEEE Transactions on Dependable and Secure Computing*. https://doi.org/10.1109/TDSC.2019.2909902

**Citing this paper**
Please note that where the full-text provided on King's Research Portal is the Author Accepted Manuscript or Post-Print version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version for pagination, volume/issue, and date of publication details. And where the final published version is provided on the Research Portal, if citing you are again advised to check the publisher's website for any subsequent corrections.

**General rights**
Copyright and moral rights for the publications made accessible in the Research Portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognize and abide by the legal requirements associated with these rights.

•Users may download and print one copy of any publication from the Research Portal for the purpose of private study or research.
•You may not further distribute the material or use it for any profit-making activity or commercial gain
•You may freely distribute the URL identifying the publication in the Research Portal

**Take down policy**
If you believe that this document breaches copyright please contact librarypure@kcl.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 04. May. 2023

# DBank: Predictive Behavioral Analysis of Recent Android Banking Trojans

Chongyang Bai, Qian Han, Ghita Mezzour, Fabio Pierazzi, and V.S. Subrahmanian

**Abstract**—Using a novel dataset of Android banking trojans (ABTs), other Android malware, and goodware, we develop the DBank system to predict whether a given Android APK is a banking trojan or not. We introduce the novel concept of a *Triadic Suspicion Graph* (TSG for short) which contains three kinds of nodes: goodware, banking trojans, and API packages. We develop a novel feature space based on two classes of scores derived from TSGs: *suspicion scores* (SUS) and *suspicion ranks* (SR)—the latter yields a family of features that generalize PageRank. While TSG features (based on SUS/SR scores) provide very high predictive accuracy on their own in predicting recent (2016-2017) ABTs, we show that the combination of TSG features with previously studied lightweight static and dynamic features in the literature yields the highest accuracy in distinguishing ABTs from goodware, while preserving the same accuracy of prior feature combinations in distinguishing ABTs from other Android malware. In particular, DBank's overall accuracy in predicting whether an APK is a banking trojan or not is up to 99.9% AUC with 0.3% false positive rate. Moreover, we have already reported two unlabeled APKs from VirusTotal (which DBank has detected as ABTs) to the Google Android Security Team—in one case, we discovered it before any of the 63 anti-virus products on VirusTotal did, and in the other case, we beat 62 of 63 anti-viruses on VirusTotal. This suggests that DBank is capable of making new discoveries in the wild before other established vendors. We also show that our novel TSG features have some interesting defensive properties as they are robust to knowledge of the training set by an adversary: even if the adversary uses 90% of our training set and uses the exact TSG features that we use, it is difficult for him to infer DBank's predictions on APKs. We additionally identify the features that best separate and characterize ABTs from goodware as well as from other Android malware. Finally, we develop a detailed data-driven analysis of five major recent ABT families: `FakeToken`, `Svpeng`, `Asacub`, `BankBot`, and `Marcher`, and identify the features that best separate them from goodware and other malware.

**Index Terms**—Android Banking Trojans, Machine Learning, Graph Models, Malware

## 1 Introduction

According to *Statista* [35], Android was the most widely used smartphone OS in the world with a market share of 87.9% at the end of the second quarter of 2017. Another Statista report [36] shows that Android had more than 3 times as many vulnerabilities reported in it as the nearest competing mobile operating system, iOS. Moreover, a Kaspersky Labs report of April 2018 specifies the existence of 94,368 mobile banking trojans [32], showing that ABTs are rapidly and dynamically evolving in their attacks. These three statistics collectively suggest that the problem of detecting Android Banking Trojans (ABTs for short) is of enormous importance.

The main goal of this paper is to address the ABT threat. We develop a framework called DBank that tackles detection and characterization of ABTs using multiple features combinations and ML algorithms. In particular, DBank relies both on *lightweight* features[1] from the prior literature, and on a novel feature space (TSG) that we show to have high accuracy and interesting properties.


- *C. Bai, Q.Han and V.S. Subrahmanian are with the Department of Computer Science and the Institute for Security, Technology, and Society, Dartmouth College, Hanover, NH 03755, USA. G. Mezzour is with the Dept. of Computer Science and Logistic and the TICLab, Universite Internationale de Rabat, Sala El Jadida, Morocco. F. Pierazzi is with King's College London and Royal Holloway, University of London, UK.*
- **Corresponding author:** *Professor V.S. Subrahmanian.*


1. The focus on lightweight features of DBank is to ensure scalable feature extraction costs, as heavyweight static and dynamic analysis can take up to 1-2 hours for each Android sample.

Most prior work on Android malware detection considered generic "malware", without focusing specifically on ABTs. Many works on banking trojans focus on the desktop or Web browser domains [12], [16], [7], which are fundamentally different to defend than a mobile environment in terms of both static [4] and dynamic [38] analysis. There are some works which do focus specifically on ABTs—however they either assume post-mortem analysis of samples already known to be ABTs (e.g., [6], [11]) or conduct surveys without proposing a detection algorithm (e.g., [15]). To the best of our knowledge, this paper proposes the first system for both detection (Section 4) and characterization of ABTs (Section 6) and analyzes recent ABTs from 2016-2017 (Section 3). We additionally propose a novel feature-space (TSG in Section 2) which we show provides both high detection accuracy (Section 4), as well as some interesting defensive properties with respect to some adversary attacks (Section 5).

This paper makes four major contributions. First, given an Android APK, we automatically predict whether it is a banking trojan or not. We show that DBank achieves this with high accuracy, even after removing isomorphic feature vectors[2], with an AUC up to 99.9% and a false positive rate of 0.3% (without isomorphic samples) to distinguish ABTs from goodware, and up to 95.3% AUC with FPR 2.7% to distinguish ABTs from other-malware. DBank achieves this via the introduction of a novel structure called a *Triadic Suspicion Graph* (TSG for short), along with two novel graph metrics called suspicion scores (SUS) and suspicion ranks (SR) that are derived from TSGs. Moreover, we present a

novel *Window-Based TSG Feature Creator*. We show that TSG-based features alone, when used in conjunction with off the shelf machine learning algorithms, generate high predictive accuracy — but when used in conjunction with additional features derived from more traditional static and dynamic analysis [30], [4], [10], [37], [13] generate even better results.

Second, we show that TSG-based features have some interesting defensive properties in the presence of adversaries who might guess and obtain even a large part of our training set. In the real world, attackers may subscribe to or have access to malware datasets (e.g., through VirusTotal). We show that even if the attackers have access to over 90% of the samples that we use, their classification accuracy will still be low. This is shown via multiple distance-based metrics as well as via a detailed Kolmogorov-Smirnov test. We additionally show that by a judicious choice of our training set from the set of openly available samples, we further compromise the ability of an adversary to reverse engineer our predictors.

Third, we conduct a thorough analysis of the features that best distinguish Android Banking Trojans (ABTs for short) from both goodware and other types of malware (e.g. ransomware, spyware, SMS fraud). In particular, we show that the following features play an important role: (i) requesting permissions to receive/modify SMS, read phone state, and control system alert windows are each highly indicative of ABTs, (ii) a low frequency of calls to some particular Android API packages (e.g., `android.widget` and `android.view`), (iii) and possible repacking activities through read, write and dynamic class load operations.

Fourth, we study 5 of the ABT families that were most prevalent in 2016 according to Kaspersky Labs [22]. In particular, we consider the following 5 ABT families: `FakeToken`, `Svpeng`, `Asacub`, `BankBot`, `Marcher`. We describe the features that best distinguish these 5 families from goodware and from other forms of malware.

The paper is organized as follows. We present our feature set in Section 2. This section primarily focuses on the novel concept of a Triadic Suspicion Graph (TSG) and the new types of metrics derived from it, namely suspicion scores and suspicion ranks. It also introduces our Window-based TSG Feature Creator methodology. We relegate a detailed description of the baseline features used to Appendix A in Online Supplementary Material because, although they are of course important, they form a baseline of lightweight static and dynamic features already studied in prior literature [30], [4], [10], [37], [13]. We then describe our Android applications dataset in Section 3, followed by our experimental results on predictive accuracy in Section 4, which also describes the key features that distinguish ABTs from both other forms of malware and from goodware. Section 5 compares the easiness of an attacker to infer predictions and features vectors under some scenarios. Section 6 then studies the 5 most prevalent forms of recent ABTs and explains how they differ not only from goodware and other malware, but also from other ABTs. Finally, Section 7 describes prior work on Android Banking Trojans, and Section 8 concludes our paper.

We conclude by noting that our DBank framework has already identified previously unknown Android banking trojans. In particular, we reported two hashes to the Google Android Security Team, both of which they confirmed. One was not identified by any of the 63 anti-virus engines running on VirusTotal at the time we reported it. The other was identified by one. Both were confirmed by Google, suggesting that DBank is not only capable of producing high AUCs and low FPRs in the lab, but is also capable to discover Android Banking Trojans in the wild.

## 2 Triadic Suspicion Graph (TSG) Features

This section describes a novel concept called a *Triadic Suspicion Graph* (TSG for short) and then shows how TSGs can be used to derive a set of novel features.

### 2.1 Triadic Suspicion Graphs (TSGs)

Given a set $\mathbb{B}$ of Android banking trojans (ABTs), a set $\mathbb{G}$ of goodware, and the set $\mathbb{A}$ of *all* available Android API packages (see the list in [14], level 23), the *Triadic Suspicion Graph* associated with $\mathbb{B}, \mathbb{G}$ denoted $TSG^{\mathbb{B},\mathbb{G}}$ is a graph with three types of vertices: members of $\mathbb{G}$, members of $\mathbb{A}$ and members of $\mathbb{B}$. A triadic suspicion graph $TSG^{\mathbb{B},\mathbb{G}}$ contains the following kinds of edges.

1) There is an edge from an ABT node $b$ to an API package $p$ if the Android app $b$ calls some method or class in the Android API package $p$ at least once.
2) There is an edge from a goodware node $g$ to an API package $p$ if the Android app $g$ calls some method or class in the Android API package $p$ at least once.
3) There is an edge from an API package $p_1$ to an API package $p_2$ if there is at least one Android API class $c_1$ imported in the package $p_1$ and at least one class $c_2$ in the package $p_2$, such that class $c_1$ imports class $c_2$.

TSGs will be very important for us because we will derive a set of features from them for each app, both ABTs and goodware.[3] It is important to note that we do *not* require the sets $\mathbb{B}, \mathbb{G}$ to be fixed. A system security analyst might use one set of reference ABTs and goodware $\mathbb{B}_1, \mathbb{G}_1$ during week 1, switch to different reference sets $\mathbb{B}_2, \mathbb{G}_2$ the next week, and keep making such changes. By doing so, they are frequently modifying the "defense surface", making it much harder for the attacker to assume or guess too much about the nature of the defense (see Section 5.2 for an evaluation of impact on performance of this mechanism). Moreover, we recommend that the sets $\mathbb{B}, \mathbb{G}$ be kept small, in order to

2. We say that two APKs (hashes) are *isomorphic* when they have identical feature vectors. In such cases, cross validation by splitting the data may cause both the training and validation sets in a given cross validation fold to contain the same feature vectors, leading to an artificial and incorrect increase in all measures of predictive accuracy. Past efforts in using machine learning in cybersecurity do not say anything about the occurrence of isomorphic samples. In this paper, we present results after removing isomorphic samples, though a removable appendix does present the results on isomorphic samples. Note that an attacker can easily generate different hashes of the same samples by first decompiling the APK, then changing only the package names or file paths in the manifest, and finally repackaging it to generate a new APK with a different hash.

3. We will apply an analogous logic to create a TSG when comparing ABTs vs. other-malware by swapping out the goodware nodes in the above definition with "other malware" nodes.

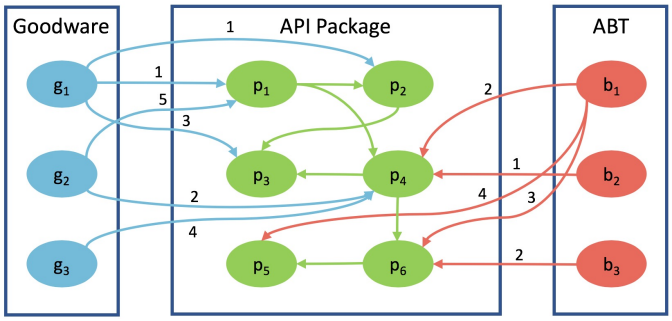



Fig. 1: A Sample Triadic Suspicion Graph (TSG).

make it difficult for the adversary to accurately guess them (see Section 5). For instance, if there are a total of 20K ABT samples available in public resources and there are 100K goodware samples, we might want to select only say 1K ABT and 1K goodware samples. It would be very hard for an adversary to guess which samples we picked for training.

Figure 1 shows a sample TSG. The red nodes are ABTs, the blue nodes represent goodware and the green nodes represent API packages. As the reader can readily observe, we see that ABT $b_2$ invokes some methods or class in API package $p_4$ but no others. On the other hand API package $p_4$ invokes classes in both $p_3$ and $p_6$, so the ABT $b_2$ indirectly invokes these API packages as well. TSGs are a special class of graphs because ABT-ABT edges, goodware-goodware edges, and ABT-goodware edges are not permitted.

We will further allow the edges in any TSG to be weighted by a *weight function* $\omega$. Eq. 1 to Eq. 5 show several weight functions, some of which we discuss below after introducing some notation. Let $f(v_i, c_j)$ be the frequency (number of times) a vertex $v_i \in \mathbb{B} \cup \mathbb{G}$ directly invokes some class in the API package $p_j$ — this can be ascertained through a straightforward static analysis of $v_i$'s code. We can then define an indicator variable as follows: $I(v_i, p_j) = 0$ if $f(v_i, p_j) = 0$, $I(v_i, p_j) = 1$ otherwise.

We now define 5 different weight functions. These represent a suite of different ways in which the relationship between an Android app $v_i$ and an Android API package $p_j$ are linked. Our goal is not to claim that these weighting mechanisms capture some ground truth, but that they represent different plausible ways to capture these relationships, which in turn will generate features to feed our machine learning models.

$$\omega_1(v_i, p_j) = f(v_i, p_j) \quad (1)$$
$$\omega_2(v_i, p_j) = f(v_i, p_j)^2 \quad (2)$$
$$\omega_3(v_i, p_j) = f(v_i, p_j)^3 \quad (3)$$
$$\omega_4(v_i, p_j) = \sqrt{f(v_i, p_j)} \quad (4)$$
$$\omega_5(v_i, p_j) = ln(f(v_i, p_j) + 1) \quad (5)$$

For instance, Equations 1, 2 and 3 can be thought of as suggesting that the relationship between an Android app and a particular Android API package is the frequency of calls to the package, the square of the frequency, and the cube of the frequency. This is because most machine learning algorithms are very sensitive to the input features and cannot always draw nonlinear inferences. Equations 4, 5 capture other possible nonlinear relationships. Again, we emphasize that we are not claiming these relationships exist, but rather that we would like to provide sufficiently diverse input features to the machine learning algorithms in DBank so that highly accurate predictions can be generated.

We use $TSG^{\mathbb{B},\mathbb{G},\omega}$ to denote the TSG generated with a set $\mathbb{B}$ of ABTs, a set $\mathbb{G}$ of goodware, and a weight function $\omega$.

## 2.2 Suspicion Scores (SUS) and Suspicion Ranks (SR)

We are now ready to define suspicion scores (SUS) and suspicion ranks (SR), which are associated with API packages in the TSG. They will constitute the basis for our novel TSG-inspired feature space.

We first present 11 definitions of SUS scores (SR is a page rank-inspired metric which we later build on top of SUS). The reason for considering multiple formulations of SUS is because we do not know a-priori which one is the best, and we do not want to overfit the model on one particular formulation. Hence, we delegate to Machine Learning the task of identifying automatically which definitions are the most important ones. In particular, we consider the following 11 SUS definitions.

$$sus(p_j) = \frac{\frac{\sum_{i=1}^{n} I(b_i,p_j)}{n}}{\frac{\sum_{i=1}^{n} I(b_i,p_j)}{n} + \frac{\sum_{i=1}^{m} I(g_i,p_j)}{m}} \quad (6)$$

$$sus(p_j) = \frac{\frac{\sum_{i=1}^{n} f(b_i,p_j)}{n}}{\frac{\sum_{i=1}^{n} f(b_i,p_j)}{n} + \frac{\sum_{i=1}^{m} f(g_i,p_j)}{m}} \quad (7)$$

$$sus(p_j) = \frac{\frac{\sum_{i=1}^{n} f(b_i,p_j)^2}{n}}{\frac{\sum_{i=1}^{n} f(b_i,p_j)^2}{n} + \frac{\sum_{i=1}^{m} f(g_i,p_j)^2}{m}} \quad (8)$$

$$sus(p_j) = \frac{\frac{\sum_{i=1}^{n} f(b_i,p_j)^3}{n}}{\frac{\sum_{i=1}^{n} f(b_i,p_j)^3}{n} + \frac{\sum_{i=1}^{m} f(g_i,p_j)^3}{m}} \quad (9)$$

$$sus(p_j) = \frac{\frac{\sum_{i=1}^{n} \sqrt{f(b_i,p_j)}}{n}}{\frac{\sum_{i=1}^{n} \sqrt{f(b_i,p_j)}}{n} + \frac{\sum_{i=1}^{m} \sqrt{f(g_i,p_j)}}{m}} \quad (10)$$

$$sus(p_j) = \frac{\frac{\sum_{i=1}^{n} \ln(f(b_i,p_j)+1)}{n}}{\frac{\sum_{i=1}^{n} \ln(f(b_i,p_j)+1)}{n} + \frac{\sum_{i=1}^{m} \ln(f(g_i,p_j)+1)}{m}} \quad (11)$$

$$sus(p_j) = \frac{\frac{\sum_{i=1}^{n} f(b_i,p_j)}{\sum_{p_j}\sum_{i=1}^{n} f(b_i,p_j)}}{\frac{\sum_{i=1}^{n} f(b_i,p_j)}{\sum_{p_j}\sum_{i=1}^{n} f(b_i,p_j)} + \frac{\sum_{i=1}^{m} f(g_i,p_j)}{\sum_{p_j}\sum_{i=1}^{m} f(g_i,p_j)}} \quad (12)$$

$$sus(p_j) = \frac{\frac{\sum_{i=1}^{n} f(b_i,p_j)^2}{\sum_{p_j}\sum_{i=1}^{n} f(b_i,p_j)^2}}{\frac{\sum_{i=1}^{n} f(b_i,p_j)^2}{\sum_{p_j}\sum_{i=1}^{n} f(b_i,p_j)^2} + \frac{\sum_{i=1}^{m} f(g_i,p_j)^2}{\sum_{p_j}\sum_{i=1}^{m} f(g_i,p_j)^2}} \quad (13)$$

$$sus(p_j) = \frac{\frac{\sum_{i=1}^{n} f(b_i,p_j)^3}{\sum_{p_j}\sum_{i=1}^{n} f(b_i,p_j)^3}}{\frac{\sum_{i=1}^{n} f(b_i,p_j)^3}{\sum_{p_j}\sum_{i=1}^{n} f(b_i,p_j)^3} + \frac{\sum_{i=1}^{m} f(g_i,p_j)^3}{\sum_{p_j}\sum_{i=1}^{m} f(g_i,p_j)^3}} \quad (14)$$

$$sus(p_j) = \frac{\frac{\sum_{i=1}^{n} \sqrt{f(b_i,p_j)}}{\sum_{p_j}\sum_{i=1}^{n} \sqrt{f(b_i,p_j)}}}{\frac{\sum_{i=1}^{n} \sqrt{f(b_i,p_j)}}{\sum_{p_j}\sum_{i=1}^{n} \sqrt{f(b_i,p_j)}} + \frac{\sum_{i=1}^{m} \sqrt{f(g_i,p_j)}}{\sum_{p_j}\sum_{i=1}^{m} \sqrt{f(g_i,p_j)}}} \quad (15)$$

$$sus(p_j) = \frac{\frac{\sum_{i=1}^{n} \ln(f(b_i,p_j)+1)}{\sum_{p_j}\sum_{i=1}^{n} \ln(f(b_i,p_j)+1)}}{\frac{\sum_{i=1}^{n} \ln(f(b_i,p_j)+1)}{\sum_{p_j}\sum_{i=1}^{n} \ln(f(b_i,p_j)+1)} + \frac{\sum_{i=1}^{m} \ln(f(g_i,p_j)+1)}{\sum_{p_j}\sum_{i=1}^{m} \ln(f(g_i,p_j)+1)}} \quad (16)$$

We discuss only a couple of them below. Let us start with Equation 6, the first suspicion scoring method. The equation says that the suspicion score of an API package $p$ is the percentage of ABTs in $\mathbb{B}$ that invoke it, divided by the sum of the percentage of ABTs in $\mathbb{B}$ that invoke it and the percentage of goodware in $\mathbb{G}$ that invoke it. On the other hand, Equation 7 says that the suspicion score of an API package $p$ is the average number of times it is called by ABTs in $\mathbb{B}$ divided by the sum of the averages of the number of times is is called by ABTs in $\mathbb{B}$ and goodware in $\mathbb{G}$.

We now adapt the definition of PageRank [29][4] to define the suspicion rank $SR(p)$ of an Android API package $p$ w.r.t. a suspicion scoring function $sus$ as:

$$SR_{sus}(p) = \frac{1-\delta}{|\mathbb{A}|} + \delta \cdot \Sigma_{p' \in \mathbb{A}, (p',p) \in E} \frac{sus(p') \cdot SR_{sus}(p')}{out(p')} \quad (17)$$

where $out(p')$ is the out-degree (number of outgoing edges) of vertex $p'$ in $TSG^{\mathbb{B},\mathbb{G},\omega}$ and $\delta \in [0,1]$ is a damping factor. As is usually done with PageRank [29], we set $\delta = 0.85$. It is important to note that unlike PageRank, suspicion rank $SR$ is not one function, but a family of 55 functions depending upon the choice of the underlying suspicion score function (one of 11 functions) and the way in which the graph is weighted (one of 5 ways). Moreover, it varies based on the choice of $\mathbb{B}, \mathbb{G}$ made by the defender which, as we have mentioned previously, should be re-vamped frequently.

*Package Call Graph.* The Android Package Call Graph (PCG for short) is the graph whose nodes are Android API package names and whose edges are defined as above: there is an edge from package $p_1$ to package $p_2$ if there is a class in $p_1$ which calls a class in $p_2$. Note that the PCG is solely dependent on the Android API [14] and is not dependent on our choice of $\mathbb{B}, \mathbb{G}$. Thus PCG is a subgraph of every Triadic Suspicion Graph and *cannot* be manipulated by the adversary because it is derived directly from the Android API, not from any specific Android app.

Prior work on related to function call graphs (e.g., dependency graphs [43] control-flow graphs [5], [27], code property graphs [40]) usually relies on abstractions of sequences of operations of specific samples. In contrast, in our TSG, the PCG is based solely on the Android OS framework dependencies, and each node of the PCG (i.e., package) is then connected to the benign and malicious applications that call it depending on APIs within their programs. In other words, while prior work computes per-app API call graphs and then creates a model or looks for similarities between different graphs, we instead create a single large TSG graph from our dataset (Figure 1), where API package edges are dependent solely on the Android framework; the SUS and SR scores are then extracted from this global TSG graph that includes also the goodware and ABT edges, and are the basis to derive TSG features. Moreover, our recommendation is to make the dataset not too large so that the adversary has difficulty identifying what we are using and that it be changed frequently, so that a "moving target" defense [19] is established.

4. PageRank is a mechanism to capture the importance of webpages using the formula $PR(v) = \frac{1-d}{N} + d \times \Sigma_{(u,v)\in E} \frac{PR(u)}{out(u)}$ where $E$ is the set of edges in the web, $N$ is the total number of nodes in the web, $d \in [0,1]$ is called the "damping factor", and $out(u)$ is the out-degree of node $u$. The $\frac{1-d}{N}$ expression captures the probability that a user will reach webpage $v$ directly (e.g. by typing it in explicitly into a browser) while the $d \times \Sigma_{(u,v)\in E} \frac{PR(u)}{out(u)}$ is intended to capture the probability of a user reaching page $v$ by following links.

### 2.3 Window-Based TSG Feature Creator

Clearly, the notions of SUS and SR give us a total of $11 + 11 = 22$ *suspicion*-based scores associated with any API package times 5 weight functions for a total of 110 possibilities. However, we need to take these scores associated with API packages and, instead, associate them with Android applications. We do this as follows.

For any of the possible suspicion scores and suspicion rank functions $\rho$, we can order all the API packages in $\mathbb{A}$ in descending order of the score returned by $\rho$. We may be tempted to think that the first API package in this descending list is the most suspicious. However, our suspicion scores and suspicion ranks are possibly *noisy*. We therefore arrange the sorted list of API packages in $\mathbb{A}$ into buckets consisting of the first $W$ API packages, the second $W$ API packages, the third $W$ API packages and so forth as shown in Figure 2. We call $W$ the *window size*.

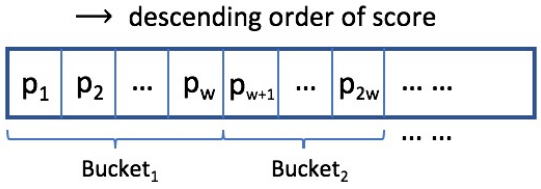



Fig. 2: Window-based API package ranking by descending SUS/SR scores.

Each of the suspicion-based features $sf$ for a given API package call ends up in one of these buckets. The idea is that different API packages in the same bucket have similar values and ranks in the sorted list according to $\rho$. We therefore try to associate with an Android APK sample $s$, aggregate features associated with the API packages in the first bucket, the second bucket, and so forth. If API packages $p_1, \ldots, p_W$ are in a bucket, then the value of suspicion-based features $sf$ for sample $s$ can be calculated via one of the following 6 methods:

1) (Binary Value) Does sample APK $s$ call a class in at least one of $p_1, \ldots, p_W$? This is a binary feature: 0 if no, 1 if yes.
2) (Num Packages) What is the total number of API packages $p_1, \ldots, p_W$, whose classes were called by $s$?
3) (FreqSum) What is the sum of the frequencies of calls made by $s$ to classes in the API packages $p_1, \ldots, p_W$?
4) (Max, Median) What is the max (and median) of the frequencies of calls made by $s$ to classes in the API packages $p_1, \ldots, p_W$?

5) (WtSum) What is the weighted sum of the frequencies of calls made by $s$ to classes in the API packages $p_1, \ldots, p_W$ where the weight used is the $sus$ score of each API package.

In the remainder of this paper, we refer to these features as *TSG features*.

To illustrate how this works, consider the small dataset with 3 ABTs and 3 goodware samples in Figure 1. Suppose the table below shows the frequency with which the ABT sample $b_1$ calls the 6 API packages shown in Figure 1.

| | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_5$ | $p_6$ |
|---|---|---|---|---|---|---|
| Freq. | 0 | 0 | 0 | 2 | 4 | 3 |

Suppose we use Equation 6 as our suspicion scoring method. In this case, the suspicion scores (after sorting in descending order) are given by the table:

| | $p_5$ | $p_6$ | $p_4$ | $p_1$ | $p_2$ | $p_3$ |
|---|---|---|---|---|---|---|
| SUS | 1 | 1 | 0.5 | 0 | 0 | 0 |

Suppose we now use $W = 3$ as the window size. In this case, there are two buckets — the first bucket has $p_5, p_6, p_4$ in it and the second has $p_1, p_2, p_3$ in it. So the feature values for ABT $b_1$ obtained from the first bucket are: Binary Value $= 1$, NumClasses $= 3$, FreqSum $= 9$, Max $= 4$, Median $= 3$, and WtSum $= 1 \times 4 + 1 \times 3 + 0.5 \times 2 = 8$. The values of these features generated by the second bucket are all 0. Of course, this is a toy example with just 3 API packages and where we consider just 6 packages in the Android API instead of all 171.

Now suppose we repeat this process with Equation 6 and Equation 17. In this case, our table of suspicion ranks after sorting:

| | $p_5$ | $p_3$ | $p_6$ | $p_1$ | $p_2$ | $p_4$ |
|---|---|---|---|---|---|---|
| SR | 0.06 | 0.04 | 0.04 | 0.03 | 0.03 | 0.03 |

So the feature values for ABT $b_1$ obtained from the first bucket are: Binary Value $= 1$, NumClasses $= 2$, FreqSum $= 4+0+3 = 7$, Max $= 4$, Median $= 3$, and WtSum $= 0.06 \times 4 + 0.04 \times 0 + 0.04 \times 3 = 0.36$. Of the API calls in the second bucket, $b_1$ only calls $p_4$ and hence its corresponding feature values are $1, 1, 2, 2, 0$ and $0.03 \times 2 = 0.06$, respectively.

The above discussion shows how *part* of the feature vector for sample $b_1$ is created. In total, we therefore have 22 suspicion scores and ranks and 6 TSG feature computations for each *window* of size $W$, leading to a total of 132 features for each window. Of a total of 171 API packages in all, this means that we will have a total of $132 \times \lfloor \frac{171}{W} \rfloor$ features in all for any given APK sample $s$. When $W = 5$, for instance, we have a total of $3,960$ features associated with any Android APK sample. (We experiment with different values of window size). These are the features that we will work with when making predictions.

**Other Features.** In addition to the TSG features (3,960 in all), we also used 130 features from the Android Manifest, 171 features for API Package, 3,459 features for API Class, and approximately 40,000 features from dynamic analysis of the code. Dynamic analysis was achieved by using the Koodous online service [1]. We do not describe the lightweight static and dynamic analysis features here, as they are reimplemented from related works [10], [37], [4]. We report more details on them in Appendix A (Online Supplementary Material).

TABLE 1: Dataset Composition

| | Goodware | ABT | Other Malware |
|---|---|---|---|
| Raw dataset | 3,535 | 7,107 | 4,478 |
| No-Isomorphic | 2,998 | 1,061 | 1,056 |

TABLE 2: Classification Results (in %) on Different Group Width (W) using TSG features.

| Dataset | W | $F_1$ | Prec. | Rec. | AUC | FPR | FNR |
|---|---|---|---|---|---|---|---|
| ABT vs. Goodware | 5 | 94.1 | 96.3 | 91.9 | **98.9** | 1.2 | 8.1 |
| ABT vs. Goodware | 10 | 93.9 | 96.1 | 91.7 | 98.6 | 1.3 | 8.3 |
| ABT vs. Goodware | 15 | 93.0 | 96.1 | 90.2 | 98.5 | 1.3 | 9.8 |
| ABT vs. Goodware | 20 | 93.1 | 95.5 | 90.8 | 98.5 | 1.5 | 9.2 |
| ABT vs. Other-malware | 5 | 85.2 | 87.6 | 82.9 | **92.4** | 11.8 | 17.1 |
| ABT vs. Other-malware | 10 | 85.1 | 87.2 | 83.1 | 92.3 | 12.3 | 16.9 |
| ABT vs. Other-malware | 15 | 84.9 | 87.1 | 82.8 | 91.7 | 12.3 | 17.2 |
| ABT vs. Other-malware | 20 | 84.4 | 85.9 | 82.9 | 91.6 | 13.6 | 17.1 |

## 3 The DBank Dataset

In this section, we briefly review the recent dataset used to evaluate the DBank system. We create our dataset by downloading Android APKs from VirusTotal during the 2016-2017 time period that were classified as ABTs by at least 5 antivirus tools. We did the same with goodware (0 detections) and with other types of Android malware (Android malware not identified as ABT, and with at least 5 antivirus detections).

Table 1 summarizes the composition of the dataset. The *raw dataset* row in this table shows the number of samples in each category that were downloaded *and* successfully dynamically executed on Koodous [1]. However, it is possible that two samples with different file hashes have the same feature vectors. In this case, we say that these two samples are *isomorphic*. Training/testing on the raw dataset would run the risk of artificially inflating prediction quality because a sample might be in the training set, while an isomorphic copy could also be in the test set. In this case, the problem of labeling the test example would be unrealistically trivial, leading to an unjustifiable increase in prediction performance. In order to avoid this, we build a *No-Isomorphic* dataset which retained only one copy of samples that had the same feature vectors.

In the remainder of the paper, we report the No-Isomorphic results as they represent a worst-case scenario for our classifier; for the sake of completeness, we also include the isomorphic results in Appendix C.

## 4 Predictive Accuracy Results

We are now ready to describe the experiments we conducted to evaluate the performance of DBank using different combinations of our novel TSG (Section 2) and traditional features. We followed the usual 10-fold cross validation protocol. In each fold, we split our data into a training set $TR$ and a test set $TS$. The training set itself was split into a partial training set $PTR$ and a validation set $PVS$. The goal was to optimize parameters using the training set alone (i.e. using $PTR$ to train and $PVS$ to validate) and then predict on the hold-out (blind) test set.

This section reports only non-isomorphic results (which represent a worst-case scenario for our DBank system). Results with isomorphic features are reported in Appendix C.

TABLE 3: AUC (%) and FPR (%) on ABT vs. Goodware with no isomorphic samples, divided into three groups. Cells with gray background highlight the best AUC in the group. We report results for the following features (and combinations of them): TSG, Manifest (M), Dynamic (D), API package (AP), and API class (AC).

| | | KNN | | LR | | DT | | NB | | RF | | GBDT | | MLP | | SVM | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Features** | AUC | FPR | AUC | FPR | AUC | FPR | AUC | FPR | AUC | FPR | AUC | FPR | AUC | FPR | AUC | FPR |
| INDIVIDUAL | M (Manifest) | 90.3 | 9.0 | 97.4 | 1.7 | 94.1 | 3.2 | 95.2 | 4.4 | 98.3 | 0.5 | 95.2 | 2.7 | 50.0 | 40.0 | 57.5 | 60.5 |
| | AP (API Package) | 94.9 | 6.2 | 96.4 | 9.8 | 94.4 | 3.4 | 91.8 | 12.1 | **98.8** | 1.0 | 97.0 | 3.0 | 84.2 | 14.3 | 83.0 | 13.8 |
| | AC (API Class) | 94.5 | 4.8 | 96.3 | 4.4 | 93.1 | 3.3 | 89.9 | 13.6 | 98.7 | 0.7 | 96.7 | 2.9 | 96.8 | 3.1 | 91.2 | 1.1 |
| | D (Dynamic) | 80.4 | 36.9 | 90.6 | 0.9 | 85.8 | 14.3 | 87.3 | 54.7 | 92.5 | 23.6 | 91.5 | 18.7 | 92.1 | 16.3 | 91.2 | 14.7 |
| | TSG (Ours) | 94.9 | 6.4 | 96.9 | 3.5 | 93.3 | 2.8 | 89.4 | 10.0 | **98.8** | 0.7 | 97.6 | 2.6 | 50.4 | 78.0 | 77.4 | 29.3 |
| COMBINED | M+D | 90.3 | 9.0 | 97.5 | 1.4 | 95.3 | 1.5 | 96.3 | 5.1 | 98.3 | 0.2 | 97.3 | 1.3 | 50.0 | 50.0 | 63.0 | 72.4 |
| | AP+M | 91.3 | 8.0 | 98.8 | 1.6 | 96.9 | 1.3 | 96.2 | 5.7 | **99.8** | 0.3 | 99.4 | 0.5 | 49.9 | 60.0 | 67.4 | 63.5 |
| | AP+D | 94.9 | 6.0 | 97.5 | 3.8 | 93.8 | 2.3 | 93.8 | 9.6 | 98.9 | 1.4 | 98.0 | 2.7 | 96.6 | 3.3 | 77.7 | 12.0 |
| | AC+M | 88.6 | 4.9 | 97.8 | 2.3 | 97.3 | 1.5 | 90.5 | 13.1 | 99.8 | 4.0 | 99.8 | 5.0 | 49.9 | 50.0 | 66.9 | 32.1 |
| | AC+D | 94.7 | 4.5 | 97.6 | 3.3 | 93.7 | 2.0 | 90.4 | 13.1 | 98.8 | 4.0 | 97.6 | 1.4 | 96.1 | 2.5 | 94.4 | 6.1 |
| | AC+AP | 94.1 | 4.7 | 96.4 | 4.4 | 93.4 | 2.8 | 89.9 | 13.6 | 98.7 | 0.4 | 97.3 | 2.0 | 92.7 | 5.6 | 87.3 | 12.3 |
| | AP+M+D | 90.8 | 8.1 | 97.8 | 2.0 | 95.4 | 1.8 | 96.6 | 6.1 | 99.1 | 0.3 | 98.2 | 1.6 | 50.0 | 70.0 | 51.8 | 62.3 |
| | AC+M+D | 88.6 | 4.9 | 97.8 | 2.2 | 97.1 | 1.3 | 91.0 | 12.5 | 99.7 | 0.2 | 99.7 | 0.5 | 50.1 | 40.0 | 60.6 | 42.5 |
| | AC+M+AP | 89.5 | 4.9 | 98.2 | 2.0 | 96.9 | 1.7 | 90.4 | 13.1 | 99.8 | 0.3 | 99.6 | 0.7 | 50.0 | 60.0 | 62.0 | 39.5 |
| | AC+AP+D | 94.2 | 4.6 | 97.6 | 3.5 | 93.6 | 2.4 | 90.4 | 12.9 | 98.9 | 0.4 | 97.3 | 1.5 | 89.8 | 3.7 | 94.1 | 4.4 |
| | AP+AC+M+D | 89.5 | 4.9 | 98.0 | 2.0 | 97.1 | 1.3 | 90.8 | 12.5 | 99.7 | 0.3 | 99.7 | 0.5 | 50.0 | 50.0 | 50.4 | 58.4 |
| COMBINED (WITH TSG) | TSG+AP | 94.9 | 6.4 | 97.0 | 3.6 | 93.4 | 2.8 | 89.3 | 10.1 | 98.8 | 0.7 | 97.3 | 2.6 | 57.8 | 78.0 | 83.2 | 29.3 |
| | TSG+M | 94.4 | 7.9 | 98.6 | 0.9 | 96.7 | 1.6 | 89.6 | 9.7 | 99.8 | 0.4 | 99.6 | 0.4 | 49.5 | 80.0 | 52.1 | 67.9 |
| | TSG+D | 94.9 | 6.4 | 94.2 | 2.2 | 93.7 | 2.1 | 89.4 | 9.8 | 99.0 | 0.7 | 97.9 | 2.4 | 61.2 | 81.0 | 69.4 | 16.3 |
| | TSG+AC | 83.2 | 8.0 | 96.2 | 5.6 | 92.8 | 3.4 | 84.0 | 22.5 | 98.3 | 2.5 | 96.6 | 2.8 | 49.9 | 0.1 | 83.1 | 17.6 |
| | TSG+M+D | 94.4 | 7.6 | 98.5 | 0.8 | 97.4 | 1.2 | 89.7 | 7.8 | 99.7 | 0.8 | 99.5 | 1.1 | 50.0 | 70.0 | 48.8 | 46.4 |
| | TSG+AP+M | 94.5 | 7.6 | 98.9 | 1.1 | 97.1 | 1.4 | 91.9 | 8.3 | 99.8 | 0.4 | 99.5 | 0.5 | 50.0 | 80.0 | 57.8 | 67.0 |
| | TSG+AP+D | 94.5 | 6.4 | 98.2 | 2.3 | 93.7 | 2.3 | 91.7 | 8.6 | 98.8 | 1.3 | 98.2 | 2.6 | 52.6 | 70.1 | 82.3 | 35.4 |
| | TSG+AC+M | 88.5 | 4.9 | 97.8 | 2.3 | 97.3 | 1.5 | 90.5 | 13.1 | 99.8 | 0.4 | 99.8 | 0.5 | 66.9 | 32.1 | 74.7 | 37.2 |
| | TSG+AC+D | 94.1 | 4.7 | 93.2 | 5.5 | 93.2 | 4.3 | 88.2 | 4.5 | 98.7 | 2.1 | 96.0 | 4.5 | 54.9 | 66.8 | 82.5 | 21.4 |
| | TSG+AC+AP | 94.2 | 4.5 | 97.3 | 3.5 | 93.2 | 2.9 | 89.6 | 14.1 | 99.1 | 2.0 | 98.4 | 2.2 | 52.3 | 50.0 | 94.8 | 5.6 |
| | TSG+AP+M+D | 94.5 | 7.8 | 98.3 | 0.9 | 97.6 | 1.0 | 89.9 | 9.7 | **99.9** | 0.3 | 99.6 | 0.6 | 49.8 | 70.0 | 64.5 | 48.9 |
| | TSG+AC+M+D | 92.7 | 3.2 | 98.7 | 2.3 | 97.0 | 1.2 | 90.0 | 14.0 | 99.5 | 1.5 | 99.8 | 0.5 | 50.1 | 80.0 | 81.1 | 19.1 |
| | TSG+AC+M+AP | 92.8 | 3.1 | 98.7 | 2.1 | 97.0 | 1.3 | 89.7 | 14.0 | 99.7 | 1.1 | 99.8 | 0.5 | 50.2 | 40.4 | 80.7 | 18.5 |
| | TSG+AC+AP+D | 94.2 | 4.5 | 98.2 | 3.0 | 92.8 | 2.4 | 89.7 | 13.8 | 99.0 | 2.3 | 98.9 | 2.1 | 58.4 | 67.2 | 81.9 | 18.1 |
| | TSG+AP+AC+M+D | 92.8 | 3.1 | 98.6 | 2.3 | 97.4 | 1.1 | 89.9 | 13.8 | 99.6 | 1.4 | 99.8 | 0.4 | 50.0 | 70.0 | 68.8 | 36.7 |

### 4.1 Identifying the best $W$

In order to identify the best possible value of window-size $W$ to build the feature vectors based on TSG (Section 2.3), we tested on $PTR$ with different values of $W$ and validated on $PVS$. Table 2 shows the results how changing the values of $W$ affects the $F_1$-Score, AUC, Precision, and Recall using the Random Forest classifier both to distinguish ABTs from Goodware and in distinguishing ABTs from Other Malware. While the differences in $F_1$-Score and AUC (the two measures that best capture classifier performance) are not huge, we see that in W=5 generally performs best. Hence, in the remainder of the experiments, we use W=5 as the window size.

### 4.2 ABT vs. Goodware

In order to evaluate DBank performance in distinguishing between ABTs and goodware, we tested a suite of 8 classifiers: $k$-Nearest Neighbor (KNN), Logistic Regression (LR), Decision Tree (DT), Naive Bayes (NB), Random Forest (RF), Gradient Boosting Decision Tree (GBDT), Multi-Layer Perceptrons (MLP) and Support Vector Machines (SVM). We also varied the set of features provided as input to the classifiers in order to understand the impact of different feature sets on performance. In particular, we examine the following feature sets: DBank's Triadic Suspicion Graph-based features (TSG, for short—our proposed feature set described in Section 2), Manifest, API package call, API class call, and Dynamic (more details in Appendix A). We consider performance for each of these feature sets alone, as well as combinations of them.

Table 3 shows the AUC and FPR of using different classifiers and feature space combinations. The table considers the "No-Isomorphic" scenarios (Section 3), where feature vectors are unique. We see that RF generates the best resulting irrespective of the feature types used, yielding up to 0.999 AUC and 0.003 FPR. In particular, when individual features are considered, our TSG performs best along API Package features. The best performance of 99.9% AUC is achieved when a combination of different features (including TSG) is used.

### 4.3 ABT vs. Other Malware

We now compare the same 8 classifiers with different feature combinations to predict whether a malicious object belongs to either the banking trojan class (i.e., ABT) or is in the "other-malware" type category. The resulting AUCs and FPRs are shown in Table 4. Again, we see that DBank with Decision Tree-based classifiers (RF without TSG features, and GBDT with TSG features) generate the best results with an AUC of 95.3% and an FPR between 2.7% and 3.4%. When considering individual features the Manifest performs best (94.1% AUC), whereas our TSG features perform slightly lower (92% AUC). Despite this slightly lower performance of TSG in this setting, we later show that TSG features have some interesting defensive properties that make it harder for the attacker to guess the defender's predictions (see Section 5).

Moreover, we observe that ABT vs. other-malware performance is slightly lower than what we saw when distinguishing ABTs from goodware, and reflects the fact that

TABLE 4: AUC (%) and FPR (%) on ABT vs. Other-malware with no isomorphic samples, divided into three groups. Cells with gray background highlight the best AUC in the group. We report results for the following features (and combinations of them): TSG, Manifest (M), Dynamic (D), API package (AP), and API class (AC).

| | | KNN | | LR | | DT | | NB | | RF | | GBDT | | MLP | | SVM | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Features | AUC | FPR | AUC | FPR | AUC | FPR | AUC | FPR | AUC | FPR | AUC | FPR | AUC | FPR | AUC | FPR |
| INDIVIDUAL | M (Manifest) | 62.8 | 26.1 | 88.8 | 13.7 | 82.9 | 6.0 | 83.5 | 14.2 | **94.1** | 4.6 | 90.8 | 6.8 | 50.0 | 50.0 | 48.6 | 57.7 |
| | AP (API Package) | 84.3 | 25.5 | 80.2 | 20.7 | 82.0 | 18.1 | 76.8 | 29.4 | 92.5 | 11.0 | 90.7 | 15.6 | 67.1 | 29.5 | 58.4 | 20.5 |
| | AC (API Class) | 83.0 | 7.8 | 81.9 | 11.9 | 80.8 | 7.1 | 77.9 | 20.2 | 91.7 | 5.9 | 89.5 | 7.7 | 80.1 | 23.5 | 71.3 | 18.0 |
| | D (Dynamic) | 72.9 | 5.8 | 81.0 | 24.8 | 76.7 | 19.8 | 82.8 | 15.3 | 83.4 | 22.0 | 82.0 | 22.6 | 83.3 | 20.5 | 84.1 | 24.1 |
| | TSG (Ours) | 84.1 | 8.6 | 85.3 | 8.7 | 80.6 | 7.6 | 75.7 | 25.4 | 92.0 | 6.1 | 89.4 | 7.5 | 54.9 | 65.9 | 63.8 | 20.4 |
| COMBINED | M+D | 62.7 | 25.8 | 90.0 | 9.8 | 84.6 | 5.9 | 88.3 | 13.1 | 94.2 | 5.1 | 90.1 | 7.9 | 49.7 | 60.0 | 49.2 | 19.9 |
| | AP+M | 62.4 | 24.1 | 89.4 | 7.6 | 82.0 | 5.5 | 88.3 | 10.6 | **95.3** | 2.7 | 95.3 | 2.8 | 50.0 | 69.9 | 54.1 | 57.9 |
| | AP+D | 83.8 | 6.9 | 85.1 | 6.7 | 81.8 | 4.8 | 87.0 | 8.6 | 93.3 | 3.4 | 91.3 | 6.3 | 77.6 | 16.6 | 69.2 | 21.6 |
| | AC+M | 61.0 | 33.7 | 85.5 | 18.4 | 80.3 | 17.3 | 82.3 | 20.6 | 94.4 | 7.8 | 92.5 | 10.5 | 50.0 | 60.0 | 53.1 | 43.8 |
| | AC+D | 83.7 | 18.9 | 84.1 | 18.3 | 81.4 | 15.7 | 81.4 | 15.0 | 92.2 | 9.6 | 90.0 | 11.8 | 82.6 | 12.7 | 73.9 | 32.5 |
| | AC+AP | 83.9 | 19.0 | 83.2 | 20.4 | 81.1 | 16.8 | 77.3 | 25.5 | 91.7 | 10.2 | 90.4 | 13.1 | 75.5 | 30.7 | 65.2 | 27.4 |
| | AP+M+D | 63.5 | 21.1 | 88.0 | 9.1 | 84.3 | 5.3 | 89.6 | 10.5 | 94.2 | 4.2 | 92.0 | 6.8 | 49.9 | 40.0 | 49.1 | 31.4 |
| | AC+M+D | 60.9 | 33.8 | 83.9 | 19.1 | 81.2 | 16.8 | 85.1 | 12.8 | 94.3 | 7.6 | 93.1 | 10.4 | 50.0 | 69.9 | 55.0 | 34.9 |
| | AC+M+AP | 61.7 | 34.0 | 84.6 | 18.7 | 81.0 | 17.4 | 81.6 | 21.2 | 94.5 | 8.2 | 93.3 | 10.6 | 50.1 | 40.0 | 55.2 | 29.1 |
| | AC+AP+D | 83.6 | 18.2 | 84.1 | 18.0 | 81.5 | 16.4 | 80.9 | 16.4 | 92.3 | 9.2 | 90.8 | 13.2 | 81.1 | 13.7 | 70.5 | 24.3 |
| | AP+AC+M+D | 61.6 | 34.0 | 84.4 | 17.7 | 82.2 | 16.6 | 84.4 | 13.6 | 94.1 | 8.2 | 93.7 | 10.8 | 49.8 | 40.1 | 54.3 | 51.2 |
| COMBINED (WITH TSG) | TSG+AP | 84.4 | 9.3 | 85.7 | 9.0 | 80.6 | 7.7 | 69.7 | 25.5 | 92.5 | 6.0 | 89.9 | 7.5 | 50.5 | 75.5 | 63.5 | 15.5 |
| | TSG+M | 64.9 | 21.4 | 90.7 | 4.6 | 82.6 | 5.7 | 76.6 | 24.8 | 94.1 | 2.9 | 94.8 | 2.8 | 50.0 | 30.0 | 54.0 | 24.1 |
| | TSG+D | 83.6 | 7.2 | 87.7 | 5.7 | 82.4 | 5.6 | 76.5 | 24.7 | 92.3 | 4.1 | 92.0 | 6.1 | 54.3 | 73.2 | 66.6 | 17.3 |
| | TSG+AC | 73.3 | 29.0 | 79.4 | 22.4 | 77.4 | 20.4 | 69.1 | 32.0 | 88.6 | 17.8 | 92.9 | 12.3 | 50.7 | 20.0 | 68.0 | 26.0 |
| | TSG+M+D | 64.9 | 21.4 | 90.7 | 4.4 | 82.8 | 5.4 | 74.4 | 24.3 | 94.0 | 3.2 | 94.6 | 2.7 | 49.9 | 80.0 | 53.7 | 53.8 |
| | TSG+AP+M | 65.0 | 21.4 | 90.2 | 4.6 | 81.1 | 5.9 | 76.6 | 24.3 | 94.3 | 3.2 | 94.6 | 2.7 | 50.0 | 80.0 | 50.8 | 53.8 |
| | TSG+AP+D | 83.7 | 7.2 | 87.2 | 5.7 | 82.8 | 5.2 | 76.6 | 24.3 | 92.2 | 4.1 | 91.5 | 6.1 | 51.8 | 71.6 | 61.5 | 15.5 |
| | TSG+AC+M | 63.5 | 31.8 | 82.2 | 20.2 | 78.9 | 19.1 | 70.9 | 30.6 | 91.4 | 14.2 | 94.3 | 9.8 | 50.0 | 29.9 | 56.4 | 31.0 |
| | TSG+AC+D | 71.3 | 27.5 | 81.9 | 21.3 | 79.5 | 18.8 | 70.8 | 29.2 | 87.7 | 18.9 | 92.4 | 11.5 | 51.5 | 2.6 | 67.3 | 35.8 |
| | TSG+AC+AP | 75.1 | 24.2 | 80.2 | 24.0 | 79.2 | 19.2 | 69.9 | 34.7 | 90.8 | 9.5 | 92.1 | 12.5 | 50.9 | 0.1 | 69.2 | 30.2 |
| | TSG+AP+M+D | 65.2 | 20.6 | 58.2 | 4.7 | 84.2 | 4.8 | 77.8 | 23.6 | 94.3 | 2.9 | **95.3** | 3.4 | 50.0 | 50.0 | 53.6 | 16.2 |
| | TSG+AC+M+D | 63.8 | 31.8 | 83.4 | 20.4 | 79.6 | 19.5 | 72.0 | 29.4 | 90.2 | 16.1 | 93.4 | 11.3 | 49.8 | 49.8 | 57.0 | 39.4 |
| | TSG+AC+M+AP | 64.0 | 32.4 | 82.7 | 20.8 | 79.8 | 18.5 | 71.4 | 30.4 | 92.4 | 12.2 | 94.6 | 10.3 | 49.9 | 30.1 | 53.5 | 43.0 |
| | TSG+AC+AP+D | 72.3 | 24.9 | 81.7 | 21.6 | 79.1 | 19.2 | 71.5 | 28.8 | 88.1 | 17.7 | 92.4 | 11.5 | 50.9 | 1.0 | 65.3 | 35.6 |
| | TSG+AP+AC+M+D | 64.0 | 32.4 | 83.6 | 20.5 | 80.9 | 17.2 | 72.6 | 28.6 | 91.1 | 13.4 | 94.2 | 9.3 | 49.9 | 30.1 | 53.4 | 50.2 |

ABTs have more in common with other malware than they have with goodware. This is not surprising. We know that some ABTs also have a spyware component. For instance, the well-known `Asacub` ABT also acts as a form of spyware.[5].

### 4.4 Random Forest Classifier Performance

We observe that the Random Forest (RF) classifier is the best classifier at distinguishing between both ABT vs. goodware and ABT vs. other malware. In both cases, we used RF trained on decision trees. The intuitive reason for this is that the RF algorithm selects many different subsets of the training data. For each subset of the training data, it trains a *different* decision tree classifier. When testing, *each* of the many different trained decision tree classifiers makes a prediction about any given binary (e.g. is it an ABT or goodware; is it an ABT or other malware) and then a majority vote is taken in RF to decide the final classification of the binary. Because RF looks at many different subsets of the training data, it avoids potential overfitting because of peculiarities of the training data. It is therefore also not surprising that it beats out the decision tree classifier that trains just once on the entire training data.

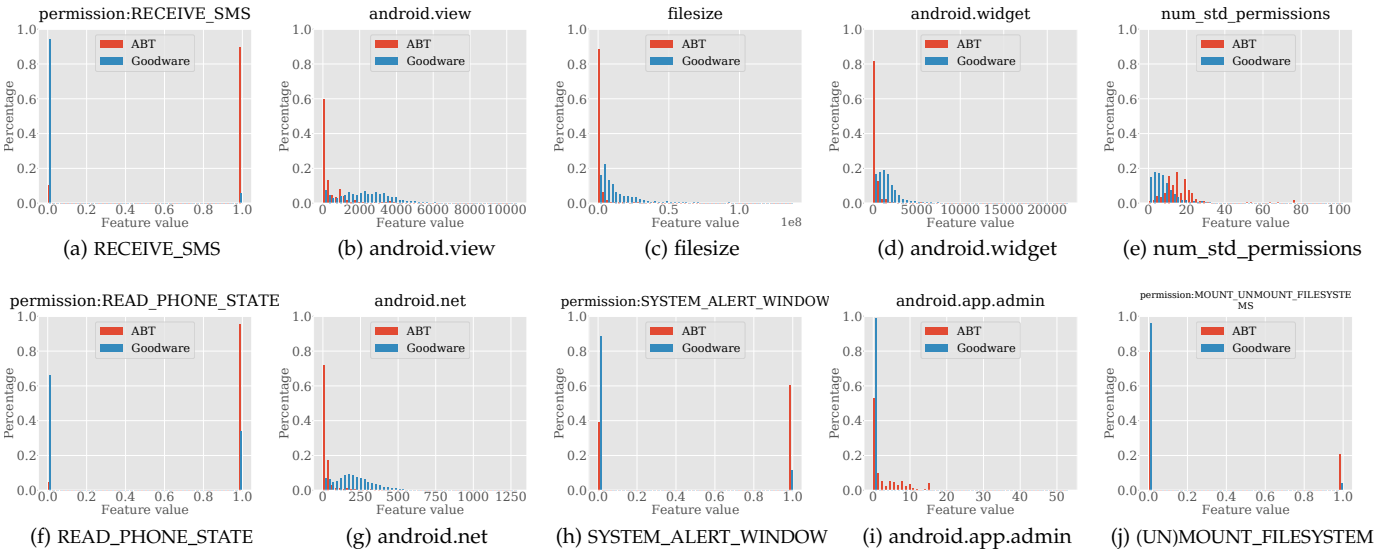
### 4.5 Identifying Key Features

As mentioned in the preceding subsections, Decision Tree (DT) and Random Forest (RF) are the best classifiers for predicting both ABTs vs. goodware and ABTs vs. other malware. The most important features of RF are determined through the average value of *Mean Decreased Impurity* (MDI) of its underlying decision trees. In particular, MDI is a standard method that progressively splits the data in two sets based on rules on individual feature values, in order to create good separations of the classes. The optimal split is identified according to the information gain concept of *Gini impurity*. Given a node $N$ in a decision tree, some set $Sat(N)$ of the samples in the training set satisfy the conditions required to reach node $N$ in the tree. The Gini score $Gini(N) = 1 - \Sigma_{\text{class}c}\mathbb{P}(c|N)^2$ where $\mathbb{P}(c|N)$ is the probability of a sample in $Sat(N)$ belonging to class $c$. A low Gini score (close to 0) means a low impurity. As we split nodes in a decision tree, the idea is to decrease the Gini score of the child nodes as compared to the parent, giving rise to the notion of mean decreased impurity (i.e. mean of decrease in Gini score). At the end of the training phase, each decision tree of the RF algorithm will have a certain MDI value for each feature, according to how relevant that feature to differentiate between the two classes; the average MDI of a feature across all the decision trees represents the importance value of that feature in the RF algorithm.

### 4.6 Key Features Distinguishing ABTs from Goodware

We also investigated the key features that distinguish ABTs from goodware in the "No-Isomorphic" case. Figure 3 shows 10 of the top 25 features that distinguish ABTs from goodware. (For space reasons, the top-25 feature histograms are reported in Appendix D). Each histogram cor-

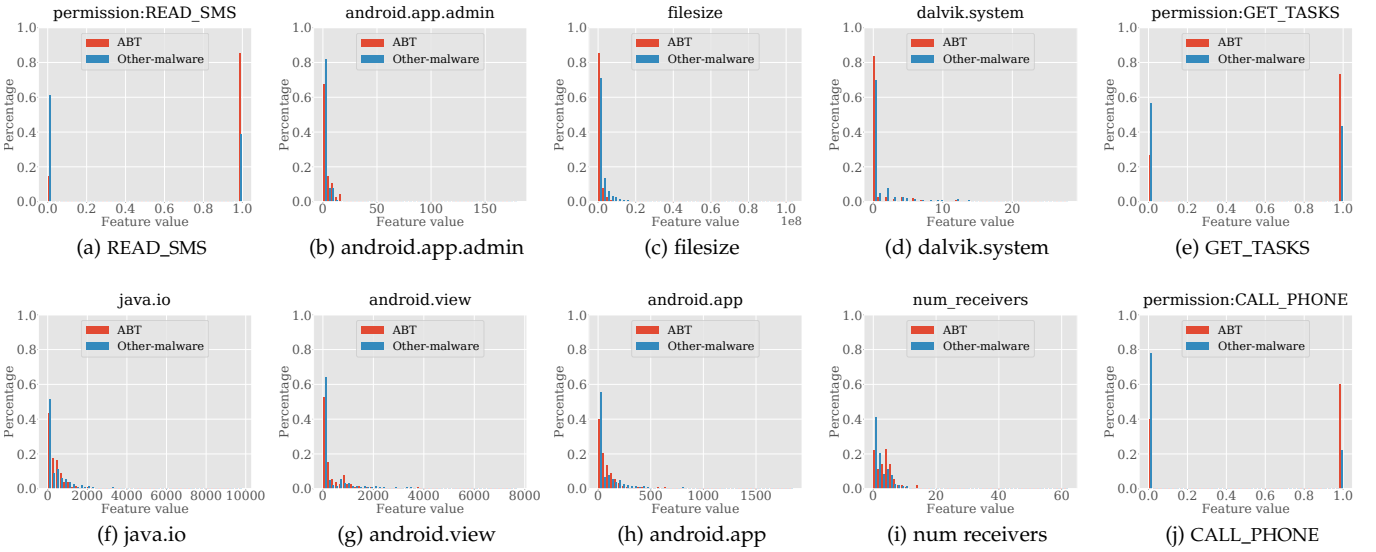
5. A Kaspersky report from April 2018 [33] states that: "We encountered the *Trojan-Banker.AndroidOS.Asacub* family for the first time in 2015, when the first versions of the malware were detected, analyzed, and found to be more adept at spying than stealing funds."




(a) RECEIVE_SMS (b) android.view (c) filesize (d) android.widget (e) num_std_permissions

(f) READ_PHONE_STATE (g) android.net (h) SYSTEM_ALERT_WINDOW (i) android.app.admin (j) (UN)MOUNT_FILESYSTEM

Fig. 3: ABT vs Goodware: Histograms of distinguishing features values.




(a) READ_SMS (b) android.app.admin (c) filesize (d) dalvik.system (e) GET_TASKS

(f) java.io (g) android.view (h) android.app (i) num receivers (j) CALL_PHONE

Fig. 4: ABT vs Other-malware: Histograms of distinguishing features values.

responds to a specific feature (e.g., number of calls to the `android.view` package); the $X$-axis reports the feature values, whereas the $Y$-axis reports the percentage of ABT (resp. goodware) that have a certain feature value. For example, Figure 3 shows that having the `RECEIVE_SMS`, `READ_PHONE_STATE` and `SYSTEM_ALERT_WINDOW` permission are some of the most important features distinguishing ABTs from goodware, because ABTs are far more likely to have this permission than goodware. Likewise, Android apps that do not invoke any methods in the API package `android.widget` and/or `android.view` are far more likely to be ABTs than goodware.

### 4.7 Key Features Distinguishing ABTs from Malware

We also investigate the key features that distinguish ABTs from other malware in the "No-Isomorphic" case. Figure 4 shows 10 of the 25 top features that distinguish ABTs from other malware. (The full set of histograms is reported in Appendix D within Online Supplementary Material). We see from Figure 4 that the `READ_SMS`, `GET_TASKS` and `CALL_PHONE` permissions are very effective in distinguishing ABTs from other malware, since ABTs are far more likely to have these permissions than other-malware. These permissions are likely used by ABTs to prevent and stop any call and alert SMS from a bank notifying unusual account activity. On the other hand, the other most important features have distributions that make using them individually for predicting whether an APK is an ABT or another malware type much more challenging. This again is not surprising as functionalities and behaviors of ABTs are harder to distinguish from other malware than from goodware. We emphasize that it is the *combination* of these features that enables us to make good predictions, and that

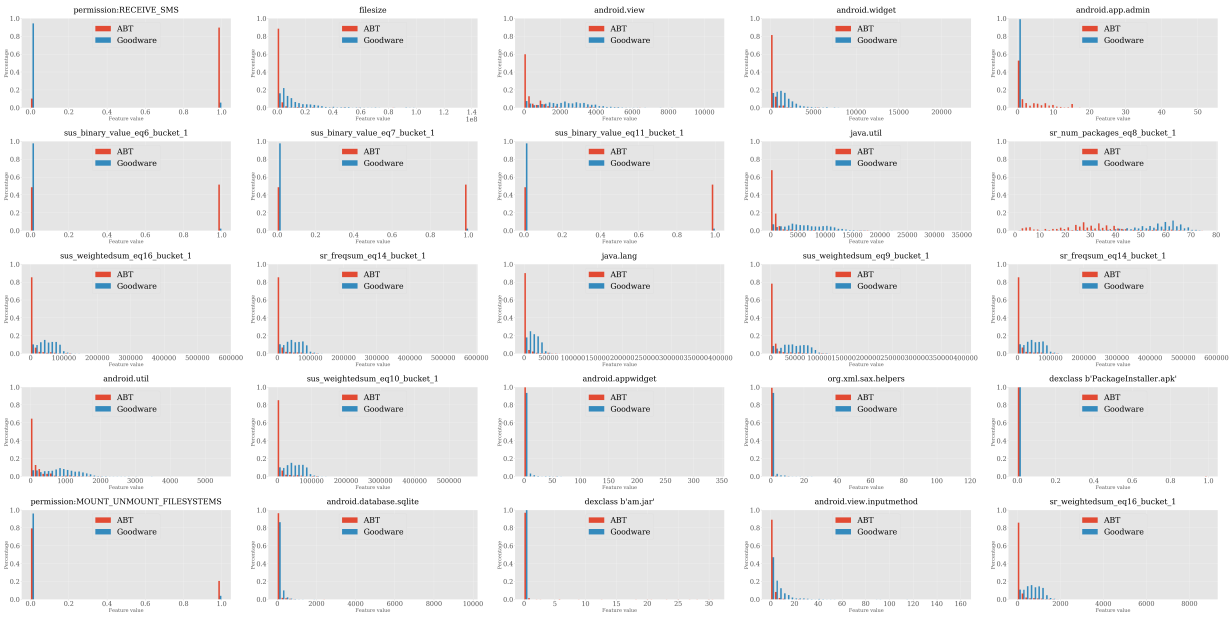



Fig. 5: Top 25 features including Suspicion Scores and Ranks.

these features by themselves do not do that.

### 4.8 Best Suspicion Scoring Methods

We also studied which suspicion scoring methods were most effective in separating ABTs from goodware. As we can see from Figure 5, 10 of the 25 most significant features involved in separating ABTs from goodware are related to suspicion scores. In particular, we see that Equations 6, 7 and 11 are the most important TSG-related features. We see immediately from Figure 5 that when the suspicion score captured by Equation 6 is close to 1 for a given binary, the probability of the binary being malware is very high. This particular suspicion scoring method captures the probability that an ABT is calling at least one of android.app.admin, javax.security.auth.callback, android.os.storage, android.app.job, android.system and android.app.usage in the 1st bucket. Likewise, we see that the tenth most important feature in separating ABTs from goodware corresponds to the number of called API packages in the 1st bucket: android.provider, java.security.cert, android.view, android.graphics, and java.nio.channels, which is ordered by Suspicion Rank computed from Equations 17 and 8.

## 5 Robustness To Some Adversary Attacks

In this section, we study the robustness of our novel TSG features against an adversary who uses machine learning using publicly available training data. Because resources such as VirusTotal are available to many people (including malicious attackers), and because large criminal networks have no trouble in gaining access to existing malware, they have access to huge numbers of Android samples. Suppose $\mathcal{S}$ is the set of all apps available to the defender which he got through some public services (e.g., VirusTotal). In particular, $\mathcal{S} \supseteq \{\mathbb{B} \cup \mathbb{G}\}$, where $\mathbb{B}$ is the set of ABTs and $\mathbb{G}$ is the set of goodware used by the defender for training.

We study two questions in this section:

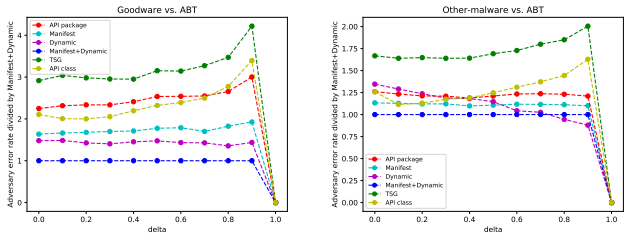
1) Suppose the attacker trains on a set that intersects part of $\mathbb{B} \cup \mathbb{G}$. How well would he infer the predictions of the defender model to craft an attack, depending on the size of this intersection? How different would the attacker feature space be?
2) How much of $\mathcal{S}$ do we need to use in a training set $\mathbb{B} \cup \mathbb{G}$ in order to ensure that we achieve and maintain high detection accuracy, while deceiving an adversary who is potentially using other subsets of $\mathcal{S}$?

The next two subsections determine the answers to these two questions.

### 5.1 Robustness based on Intersection of Defender and Adversary Training Data

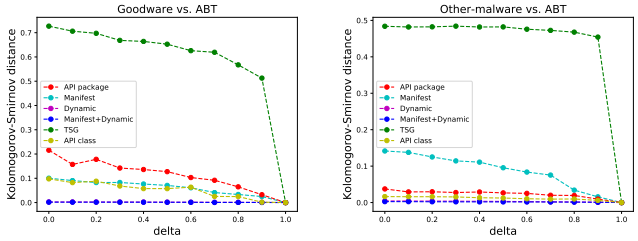
We study the first question by varying the size of the intersection $\Delta \in \{10\%, 20\%, \ldots, 90\%\}$ of the training set used by the attacker with the training set $\mathbb{B} \cup \mathbb{G}$ used by the defender. We call $\Delta$ the *overlap ratio*, which represents the percentage of data shared by both the attacker and the defender used to train their models. We define the *adversary's error rate* as the percentage of attacker predictions disagreeing with the defender model (e.g., the defender predicts a sample as ABT whereas the adversary predicts it as goodware).

We randomly select 50 subsets of samples for the adversary with similar distributions as $\mathbb{B} \cup \mathbb{G}$, and measure the ratio of the *adversary's error rate*, normalized by dividing it by the error rate of the baseline (consisting of static features from the APK's Manifest and the Dynamic features) as the overlap ratio is varied ($X$-axis). The result is shown in Figure 6: both when predicting ABTs vs. goodware, as well as predicting ABTs vs. other malware. In both cases, we see that even if the adversary knows 80% of the set $\mathbb{B} \cup \mathbb{G}$ used by us, it is still the case that the error rate generated



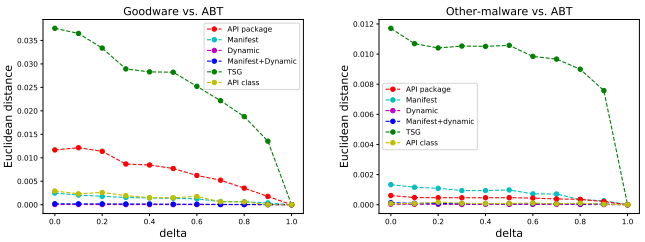

(a) Goodware vs ABT (b) Other-malware vs ABT

Fig. 6: Adversary Error Rate vs. Overlap Ratio divided by baseline: Manifest + Dynamic. Higher values imply better robustness.




(a) Goodware vs. ABT (b) Other-malware vs. ABT

Fig. 8: Kolomogorov-Smirnov distance among attacker's and defender's feature sets centroids. High values imply higher robustness.




(a) Goodware vs. ABT (b) Other-malware vs. ABT

Fig. 7: Euclidean distance among attacker's and defender's feature sets centroids. High values imply higher robustness.

using DBank is over 3 times the error of the baseline when distinguishing ABTs from goodware, and over 1.6 times the error rate of the baseline when distinguishing ABTs from other malware.

We now also evaluate how close the feature space of the attacker is to the feature space of the defender, by varying $\Delta$ as before. But this time we study the distances between the feature vectors of defender's samples using $\mathbb{B}, \mathbb{G}$ and using the samples used by the adversary. We tested our algorithm against many distance functions including Euclidean distance, Manhattan distance, Kolomogorov-Smirnov distance, Chebyshev distance and Cosine distance. We report results with Euclidean distance and K-S distance in the main body of the paper, while charts for the other distance metrics are reported in Appendix D (Online Supplementary Material).

Figure 7 and Figure 8 show the distances between the feature vectors generated using $\mathbb{B}, \mathbb{G}$ as compared to the training samples used by the adversary decreases as we vary $\Delta$. We see that the TSG features generate the biggest distances, substantially more than traditional features. In all cases, we note that as $\Delta$ increases, the distance between the feature vectors generated using $\mathbb{B}, \mathbb{G}$ as compared to the training samples used by the adversary decreases. This is not surprising as an increase in $\Delta$ means that the adversary more accurately guessed what we used to train on.

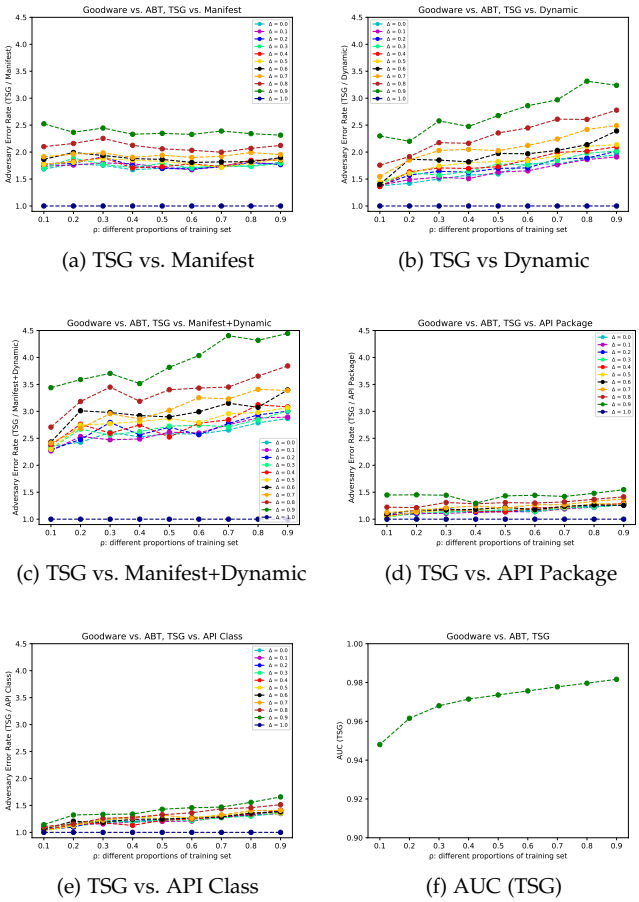
### 5.2 Accuracy-Robustness Trade-off of DBank's TSG vs traditional features

In this section, we answer the second question posed at the beginning of this section. Specifically, we would like to understand whether the defender can use a subset of his training samples, and change it over time to mislead the attacker. The first concern when considering a smaller training sample is that predictive accuracy will drop. In this section, we evaluate how AUC and robustness vary when the defender uses smaller percentages $\rho$ of the training data. Each experiment also varies the overlap ratio $\Delta$, as before. The result is shown in Figure 9: both when predicting ABTs vs. goodware. The $x$-axis varies the percentage of defender's training data $\rho$, while the $y$-axis reports values normalized by the value obtained with traditional features. By varying the size of the intersection *overlap ratio* $\Delta \in \{10\%, 20\%, \ldots, 90\%\}$ with $\mathbb{B} \cup \mathbb{G}$, also vary the actually used size of the total training sets with $\rho \in \{10\%, 20\%, \ldots, 90\%\}$. Again, we randomly select 50 sets of samples for the adversary with similar distributions as $\mathbb{B} \cup \mathbb{G}$, and measure the ratio of the *prediction's error rate*, normalized by dividing it by the error rate of the baseline (consisting of API package call features, static features from the APK's Manifest and the Dynamic features) as the overlap ratio (different non-red $+$ lines) and the training set ratio is varied ($X$-axis), and the predicted AUC results by only TSG features (red $+$ lines).

Values higher than 1 in the $y$-axis of Figure 9 imply that TSG-based features are better than traditional features (e.g., Manifest, Dynamic) in terms of adversary error rate. We see that the error rate generated using TSG features is always greater than the error of the baseline when distinguishing ABTs from goodware, while our TSG features yields high predictive accuracy AUCs. We also report AUC and show that it remains high in the different scenarios even when 20% or 30% of the training set is used. This allows the defender to use a moving defense surface [19], [20] by changing the specific training set over time while maintaining good predictive accuracy (AUC) performance.

### 5.3 Robustness against Fake Calls

Another attack that can be launched against DBank is that of "fake calls". For instance, we see from previous discussion that a low frequency of calls to some particular Android API packages may help DBank identify a binary as an ABT. An attacker can try to evade this by making more calls to that API to avoid suspicion. A fake call to an API package adds edges to the Triadic Suspicion Graph. Suppose we define the "fake call percentage" or FCP to be the ratio of the number of fake calls in a TSG to the total number of edges in the TSG




(a) TSG vs. Manifest (b) TSG vs Dynamic

(c) TSG vs. Manifest+Dynamic (d) TSG vs. API Package

(e) TSG vs. API Class (f) AUC (TSG)

Fig. 9: AUC performance and adversary error rate of DBank's TSG vs. traditional features using different proportions $\rho$ of the training set.

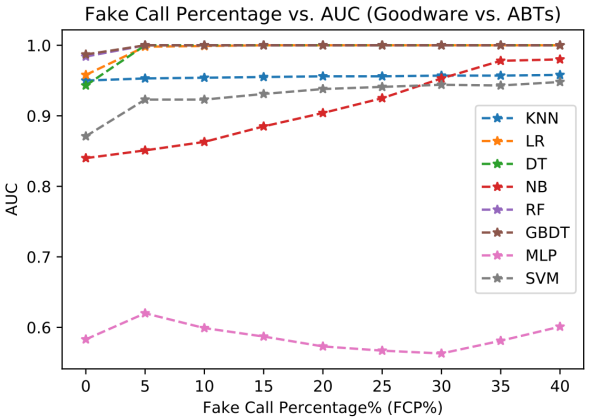



Fig. 10: Impact of Fake Call Attack on DBank's Performance

(both real and fake). Figure 10 shows that DBank is robust to an attack that tries to increase the FCP with the Random Forest classifier used in DBank showing AUCs close to 1.

## 6 Analysis of 5 Major ABT Families

In this section, we report on a detailed study and comparison of the behaviors that distinguish 5 major ABT families from both goodware and other malware. In particular, we study some five of the top banking families according to Kaspersky [22]: `FakeToken`, `Svpeng`, `Asacub`, `BankBot`, and `Marcher`.

**Note.** For space reasons, we report only a selection of the top features of these families in the main text. In particular, we report 4 top features for each ABT family vs. goodware, and 2 top features for each ABT family vs. other-malware. We show more top features for ABT vs. goodware because they offer more diversities, and are more relevant from a security analyst's perspective. The full pair of top-10 histograms for each family are reported in Appendix D).

### 6.1 Analysis of FakeToken

A major ABT family is `FakeToken`, which performs a number of malicious acts. It can capture the content of phone calls, overlay fake screens (e.g. bank screens) so that it can capture banking information, and even overlay taxi and ride-sharing apps with fake requests for credit card information—which of course is then immediately stolen [21]. In fact, security companies have estimated that `FakeToken` can overlap over 2,000 financial apps with fake screens designed to steal financial and other information [34]. Other versions of `FakeToken` also incorporate ransomware style behavior.

Figure 11(a) shows four of the top-10 features that best distinguish `FakeToken` from goodware. We specifically see that certain features individually do a great job in distinguishing `FakeToken` from goodware. In particular, unlike goodware `FakeToken` samples:

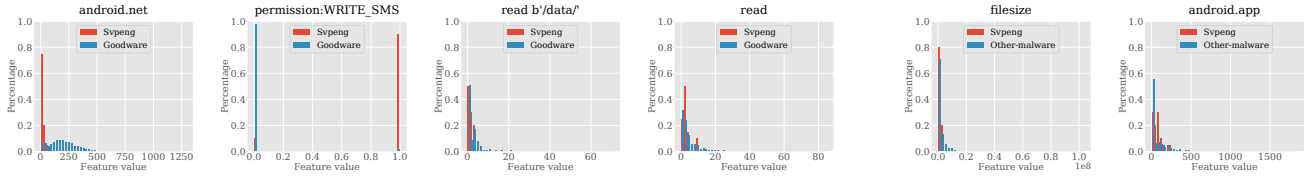
- request some peculiar permissions with much higher prevalence then goodware: permission to send SMS, permission to read phone state, and permission to get active tasks in the phone; the latter two are likely used to determine when is the best moment to act, and possibly also to monitor whether a banking app task is running.
- in general, it requests more permissions than usual goodware.
- make almost no calls to certain standard API packages such as `android.text` and `android.content` while goodware typically make a reasonable number of such calls.
- `FakeToken`'s size is smaller than most goodware, suggesting that it probably does *not* piggyback applications.

As expected, the features that distinguish `FakeToken` from other malware are different than those mentioned above and two of them are shown in Figure 11(b). In particular, unlike other-malware, `FakeToken` samples:

- tend to have a higher number of activities, which represent the interfaces shown to the user;
- in some cases calls function from `android.app.admin` package, probably as an attempt to perform administrative operations;
- have on average more *intents*, used for inter-component communications in the Android OS;
- have a different distribution in the usage of some system API packages with respect to traditional malware.


(a) `FakeToken` vs. goodware (b) `FakeToken` vs. other-malware

Fig. 11: `FakeToken` distinguishing features: Histograms of feature values.




(a) `Svpeng` vs. goodware (b) `Svpeng` vs. other-malware

Fig. 12: `Svpeng` distinguishing features: Histograms of feature values.

### 6.2 Analysis of Svpeng

According to a 2017 Kaspersky Lab report by renowned cybersecurity expert Roman Unuchek [39], `Svpeng` used phishing attacks to target users. Later, when those users went to bank websites, they were redirected to fake (but near identical) websites from which their banking credentials were stolen. Later "upgraded" versions of `Svpeng` used a Chrome browser vulnerability and also used other techniques. Other reports (e.g., [42]) describe the distribution of `Svpeng` through fake websites.

Figure 12(a) shows four of the top 10 features that best distinguish `Svpeng` from goodware. In particular, unlike goodware, `Svpeng` samples:

- request the permission to `WRITE_SMS`, which is very uncommon in goodware applications.
- make almost no calls to standard API packages such as `android.app`, `android.content`, `java.lang` and `android.net` while goodware typically make a reasonable number of such calls. This tends to suggest that `Svpeng` may not be using piggybacking of larger applications (i.e., it is not a modified version of a goodware application with added malicious functionality).
- perform a number of dynamic read and class loading operations similar to the distribution of goodware; however, there are some particular permissions (e.g., `INSTALL_LOCATION_PROVIDER`, which is usually prohibited for third-party apps and allows an app to install a custom location provider for the Location Manager) and some class names loaded at runtime (e.g., `com.suimeng-1.apk`) which allow for detection of `Svpeng` samples.

Two of the top-10 features that distinguish `Svpeng` from other-malware are shown in Figure 12(b). There is more similarity between the `Svpeng` and the other-malware features than `Svpeng` and goodware. However, there are still some distinguishing traits. In particular, `Svpeng` samples tend to use more `java.lang`, `android.widget` and `java.util` calls than other-malware. Moreover, `Svpeng` samples tend to have smaller filesize than that of other malware.

### 6.3 Analysis of Asacub

Though around since 2015, `Asacub` was the most widely spread mobile banking trojan of 2017 according to Kaspersky Labs [33]. It infects users through a phishing message injected via an SMS offering to show certain photos to users—when the users attempt to view the photo, they are infected. The attacks are widespread and estimated to target 40,000 users per day [18].

Figure 13(a) shows four of the top 10 features that best distinguish `Asacub` from goodware. In particular, `Asacub` samples:

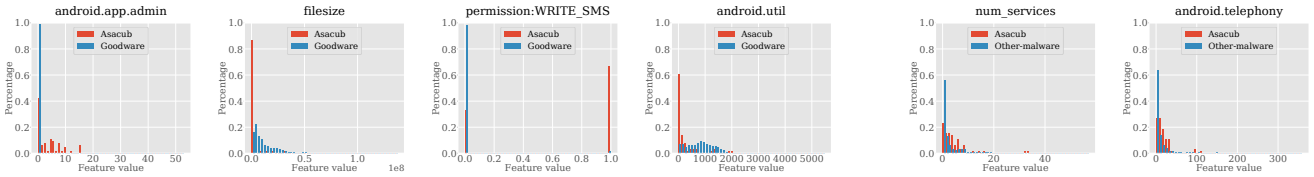
- request permissions to write SMS, which is very rare for goodware to do.
- make almost no calls to certain standard API packages such as `android.database`, `android.graphics` and `android.animation` while goodware typically uses them.
- have small filesize, suggesting that piggybacking is likely not adopted.

Figure 13(b) shows two of the top-10 features of `Asacub` vs. other-malware. In particular, unlike other malware, `Asacub` samples:

- define many background *services*, whereas other malware has more variance but mostly lower number of definitions; these may be used to carry silent malicious activities; `Asacub` samples also define a higher number of *intents* for inter-process communication;
- tend to define a higher number of Android Activities, which represent the window interfaces that could be shown to a user, and is less typical for other malware;
- use more `android.telephony` APIs, used also to interact with SMS and calls—possibly to prevent alerts and warnings from the bank.

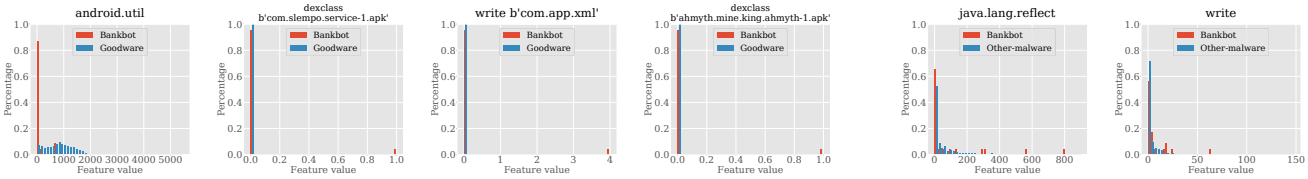
### 6.4 Analysis of BankBot

`BankBot` is another major Android malware that, like those mentioned above, tries to overlay fake screens/pages when the user attempts to visit various banking sites [23]. It is




(a) `Asacub` vs. goodware (b) `Asacub` vs. other-malware

Fig. 13: `Asacub` distinguishing features: Histograms of feature values.




(a) `BankBot` vs. goodware (b) `BankBot` vs. other-malware

Fig. 14: `BankBot` distinguishing features: Histograms of feature values.

injected into users' phones by pretending to be a normal benign app and being careful to exhibit malicious behavior not immediately upon infecting a smartphone, but after a latency period (e.g., logic time bombs).

Figure 14(a) shows four of the top-10 features of `BankBot` vs goodware. In particular, `BankBot` samples:

- perform much dynamic class loading (`dexclass`) and read/write operations on specific JAR and APK files, which possibly suggest dynamic unpacking—typically used to load code at runtime dynamically.
- Unlike goodware, they request the permission to `WRITE_SMS`.
- Perform fewer calls to methods in some APIs packages such as `java.nio` and `android.util`.

Figure 14(b) shows two of the top-10 features that distinguish `BankBot` vs other-malware. In particular, unlike other malware, `BankBot` samples:

- tend to have smaller filesize;
- perform more dynamic write operations;
- in some cases, use more `android.content`, `java.lang` and `java.reflect` APIs; in particular, the last one involves *reflection* operations (e.g., runtime resolution of function calls) typically used for obfuscation purposes and possibly associated with repacking.

### 6.5 Analysis of Marcher

the `Marcher` malware infects users by telling them that their Flash players need to be updated—once the unsuspecting users do so, their machine is infected, enabling the attacker to install apps of his own choosing as well as to nullify existing security mechanisms [44].

Figure 15(a) shows 4 of the top-10 features of `Marcher` vs goodware. In particular, `Marcher` samples:

- invoke the `android.app.admin` class 10+ times, to attempt usage of Android APIs with admin privileges.
- As other banking trojans, `Marcher` has small filesize and, unlike goodware, almost no call to some standard Android libraries.
- as in other banking trojans, `Marcher` requests much more permissions than standard goodware, among which the permission to send SMS.

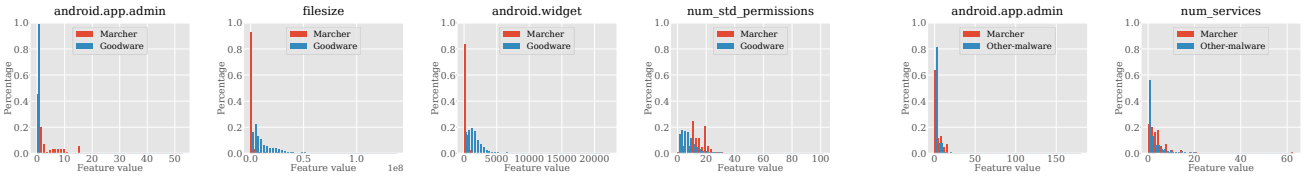
Figure 15(b) reports 2 of the top-10 features of `Marcher` vs other-malware. In particular, unlike other malware, `Marcher` samples:

- invoke methods in the Android API package `android.app.admin` package, which is slightly less common for other malware to do;
- generally define more background services, activities and intents than the average other malware;
- have smaller filesize and different distribution of API usage from the different packages.

## 7 Prior Work

We discuss 3 types of related work: literature on Android banking trojans (ABTs) which constitute the body of work closest to this paper, literature on desktop banking trojans (DBTs) and literature on Android malware detection.

**Literature on Android Banking Trojans.** The closest work to our paper is DroydSeuss [11], a tool to analyze ABTs using static and dynamic analysis to identify possible malware artifacts such as CnC domain names with frequent itemset mining, to unveil malware campaigns using the same infrastructure. DroydSeuss analyzes 4,293 Android malware samples belonging to 5 families: ZitMo, SpitMo, CitMo, iBanking and FankeBank. The major difference from our paper is that DroydSeuss is an analysis tool that takes as input applications already known to be banking trojans—it does not perform detection of ABTs from goodware and other malware; DroydSeuss does not propose a novel feature space, and it does not evaluate robustness of their methods against any adversary; furthermore, it relies on a dataset with just 5 ABT families, while our datasets contains 104 ABT families (with 34 families containing at least 10 samples). Stamba [6] provides a strategy to test Android




(a) `Marcher` vs. goodware (b) `Marcher` vs. other-malware

Fig. 15: `Marcher` distinguishing features: Histograms of feature values.

mobile apps at the application level, the communication level and the device level.The testing strategy includes 4 parts: static analysis, dynamic analysis, web app server security and device forensic. The paper contains a case study of using the strategy of a few APKs that are known to be malicious. Unlike our paper, Stamba does not use machine learning to distinguish between bankers and other malware and goodware. Actually, the strategy of used was only tested on a few APKs that are known to be malicious, unlike our paper that uses thousands of APKs. Finally, Etaher et al. [15] report on the trends and development of Zeus by reviewing papers and reports about this major banking trojan. Unlike our paper, the analysis of [15] is based on literature review and not on analyzing APKs.

**Literature on Desktop Banking Trojans.** The literature also contains papers on desktop banking trojans. The methods used in these papers do not necessarily apply to Android trojans as the the Android OS presents very different characteristics and environment for which new tools are required [38], [26]. Criscione et al. present Zarathustra [12], a tool to detect web-inject based trojans that leverages the fact that web pages are rendered differently on clean machines and on infected machines. More specifically, the paper leverages the fact that modern banking trojans are equiped with WebInject, a functionality that can silently modify a web page on the victim. Therefore, Zarathustra finds out that an APK is banking trojan if installing that APK on a host causes some web pages to be rendered differently. An evaluation of Zarathustra against 213 URLs of banking websites and 56 samples of Zeus found that the tool had no false negatives and 1% false positives. Banksafe [7] detects banking trojan infections from inside the web browser by detecting attempts made by malicious software to manipulate the browsers' networking libraries. Banksafe was tested on 1,093 banker samples from 7 families (Zeus, SpyEye, Patcher, Carberp, Silentbanker, Bebloh, Gozi and Katusha) and was able to detect 99.6% of these samples. Zarathustra and Banksafe are very different from our paper in the sense that they do not use machine learning nor static and dynamic analysis of APKs. Finally, Gregio et al. [16] develop BanDIT, a dynamic analysis system that identifies behavior related to ABTs that combines visual analysis, network traffic pattern matching and filesystem monitoring. BanDIT is a Windows kernel driver that traces the execution of an executable and the processes it interacts with. The system was able to identify 98.8% of ABTs in a manually labeled set of 1,500 malware samples. The identified ABTs were characterized according to whether they perform the following behaviors: information stealing, PAC loading, email sending, bank images and host changing. These operations may also be performed in support of pivoting activities [3]. BanDIT [16] also reported the compromised IP and email addresses found. In the DBank paper, we do not develop a novel dynamic analysis tool. Instead we use the results of dynamic and static analysis as features of machine learning classifier.

**Literature on Android Malware Detection.** Traditional work on Android malware analysis has two main objectives: *malware detection* and *family identification*. The first problem investigates whether a given Android application is benign or malicious, and the second problem tries to identify whether a given malware sample belongs to a known family (i.e., group of malware variants) so to apply same patching and removal techniques [10]. Some examples of malware detection solutions are DREBIN [4], DroidAPIMiner [2], CrowDroid [8], DroidScope [41], MARVIN [25], MaMaDroid [27],SigPID [24], MADAM [31], the deep android malware detection system [28] Hindroid [17] and [9]. DREBIN [4] and DroidAPIMiner [2] are lightweight detection methods for Android malware that uses static analysis. DREBIN detects 94 % of the malware with a low false positive rate whereas DroidAPIMiner achieves 99 % accuracy and 2.2 % false positive rate. CrowDroid [8] detects applications that have a benign name and version, but that are malicious. CrowDroid does so through a framework that collects and compares execution traces from users using a crowdsourcing approach. DroidScope [41] is a virtualization-based platform for analyzing APKs that collects native and Dalvik instruction traces and profile API-level activity. DroidScope tracks also information leakage through taint analysis. MARVIN [25] combines static and dynamic analysis and uses machine learning to distinguish between malware and goodware. MaMaDroid [27] builds a classifier to detect malware based on features extracted from a behavioral model in the form of a Markov chain, built based on the sequence of abstracted API calls performed by an app. SigPID [24] develops methods to identify significant permissions linked to the malware vs. goodware prediction problem and show that only 22 out of 135 permissions are neded to ito achieve over 90% detection accuracy. The MADAM system [31] delivers methods to identify over 96% of malicious Android apps while providing a low false positive rate. The deep android malware detection system [28] uses convolutional neural nets in conjunction with static analysis methods to separate Android malware from goodware. The Hindroid system [17] develops a heterogeneous information network and then uses metapath analysis to predict if a sample is goodware or malware. Finally, [9] develops a regression model based on decompiled code analysis for distinguishing goodware and malware.

Such works on malware detection are different from our paper for two main reasons: these works consider all malware categories as one big group whereas we focus on ABTs; moreover, these works do not perform a data-driven characterization of any particular malware category—especially ABTs, which are the focus of our paper.

## 8 Conclusion

We have presented DBank a novel framework for distinguishing between Android banking trojans (ABTs) and goodware, and other types of malware. In particular, we propose a feature set based on the novel concept of Triadic Suspicion Graph (TSG). We show that, while we achieve similar accuracy to lightweight feature sets in past work, TSG-based features are more robust to some adversary attacks, and still achieves high accuracy even when using a subset of training data. We evaluate our system on recent (2016-2017) Android ABTs, and we show how DBank can automatically extract relevant features that can highlight differences from specific ABT families vs. goodware and other-malware.

## Acknowledgements

We thank the anonymous referees for their excellent comments and suggestions. We additionally thank Salvador Mandujano and Sai Deep Tetali of the Google Android Security Team for excellent suggestions and comments on our work. Parts of the work were funded by: US Army Research Office grant W911NF1410358, ONR grants N00014-18-1-2670, N00014-16-12896, and N000141612739; NATO Science for Peace and Security grant SPS G5319; UK EPSRC grant EP/L022710/2.

## References

[1] Koodous. https://koodous.com/. Accessed: November 2018.

[2] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-level features for robust malware detection in android. In *SecureComm*. Springer, 2013.

[3] G. Apruzzese, F. Pierazzi, M. Colajanni, and M. Marchetti. Detection and threat prioritization of pivoting attacks in large networks. *IEEE Transactions on Emerging Topics in Computing*, 2017.

[4] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *NDSS*, 2014.

[5] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm SIGPLAN Notices*, 2014.

[6] S. Bojjagani and V. N. Sastry. STAMBA: Security Testing for Android Mobile Banking Apps. In S. M. Thampi, S. Bandyopadhyay, S. Krishnan, K.-C. Li, S. Mosin, and M. Ma, editors, *Advances in Signal Processing and Intelligent Recognition Systems*, volume 425, pages 671–683. Springer International Publishing, Cham, 2016. DOI: 10.1007/978-3-319-28658-7_57.

[7] A. Buescher, F. Leder, and T. Siebert. Banksafe Information Stealer Detection Inside the Web Browser. In R. Sommer, D. Balzarotti, and G. Maier, editors, *Recent Advances in Intrusion Detection*, volume 6961, pages 262–280. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. DOI: 10.1007/978-3-642-23644-0_14.

[8] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *SPSM*. ACM, 2011.

[9] L. Cen, C. S. Gates, L. Si, and N. Li. A probabilistic discriminative model for android malware detection with decompiled source code. *IEEE Transactions on Dependable and Secure Computing*, 12(4):400–412, 2015.

[10] T. Chakraborty, F. Pierazzi, and V. S. Subrahmanian. Ec2: Ensemble clustering and classification for predicting android malware families. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[11] A. Coletta, V. van der Veen, and F. Maggi. DroydSeuss: A Mobile Banking Trojan Tracker (Short Paper). In J. Grossklags and B. Preneel, editors, *Financial Cryptography and Data Security*, volume 9603, pages 250–259. Springer Berlin Heidelberg, Berlin, Heidelberg, 2017. DOI: 10.1007/978-3-662-54970-4_14.

[12] C. Criscione, F. Bosatelli, S. Zanero, and F. Maggi. ZARATHUSTRA: Extracting Webinject signatures from banking trojans. pages 139–148. IEEE, July 2014.

[13] S. K. Dash, G. Suarez-Tangil, S. Khan, K. Tam, M. Ahmadi, J. Kinder, and L. Cavallaro. Droidscribe: Classifying android malware based on runtime behavior. In *2016 IEEE Security and Privacy Workshops (SPW)*, pages 252–261. IEEE, 2016.

[14] A. P. Documentation. Api package list. https://developer.android.com/reference/packages, Visited in Mar. 2019.

[15] N. Etaher, G. R. Weir, and M. Alazab. From ZeuS to Zitmo: Trends in Banking Malware. IEEE TrustCom, 2015.

[16] A. R. A. Grégio, D. S. Fernandes, V. M. Afonso, P. L. de Geus, V. F. Martins, and M. Jino. An empirical analysis of malicious internet banking software behavior. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1830–1835. ACM, 2013.

[17] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu. Hindroid: An intelligent android malware detection system based on structured heterogeneous information network. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1507–1515. ACM, 2017.

[18] ITWEB. https://www.itweb.co.za/content/j5alr7QljxGvpYQk, Visited in Nov. 2018.

[19] S. Jajodia, A. K. Ghosh, V. Subrahmanian, V. Swarup, C. Wang, and X. S. Wang. *Moving Target Defense II: Application of Game Theory and Adversarial Modeling*, volume 100. Springer, 2012.

[20] S. Jajodia, N. Park, F. Pierazzi, A. Pugliese, E. Serra, G. I. Simari, and V. Subrahmanian. A probabilistic logic of cyber deception. *IEEE Transactions on Information Forensics and Security*, 2017.

[21] Kaspersky. FakeToken report. https://www.kaspersky.com/blog/faketoken-trojan-taxi/18002/, Visited in Nov. 2018.

[22] Kaspersky. Most prevalent Android banking trojan families in 2016. https://cdn.press.kaspersky.com/files/2017/05/Kaspersky_Lab_financial_cyberthreats_in_2016_final.pdf, Visited in Nov. 2018.

[23] M. Kumar. https://thehackernews.com/2017/11/bankbot-android-malware.html, Visited in Nov. 2018.

[24] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-an, and H. Ye. Significant permission identification for machine-learning-based android malware detection. *IEEE Transactions on Industrial Informatics*, 14(7):3216–3225, 2018.

[25] M. Lindorfer, M. Neugschwandtner, and C. Platzer. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *IEEE COMPSAC*, 2015.

[26] M. Marchetti, F. Pierazzi, A. Guido, and M. Colajanni. Countering Advanced Persistent Threats through security intelligence and big data analytics. In *2016 8th International Conference on Cyber Conflict (CyCon)*. IEEE, 2016.

[27] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini. MaMaDroid: Detecting android malware by building markov chains of behavioral models. *NDSS*, 2017.

[28] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupe, et al. Deep android malware detection. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 301–308. ACM, 2017.

[29] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[30] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In *DIMVA*. Springer, 2008.

[31] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli. Madam: Effective and efficient behavior-based android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing*, 15(1):83–97, 2018.

[32] SecureList. https://securelist.com/mobile-malware-review-2017/84139/, Visited in Nov. 2018.

[33] Securelist. Asacub report. https://securelist.com/the-rise-of-mobile-banker-asacub/87591/, Visited in Nov. 2018.

[34] Securelist. FakeToken report. https://securelist.com/booking-a-taxi-for-faketoken/81457/, Visited in Nov. 2018.

[35] Statista. https://www.statista.com/statistics/271774/share-of-android-platforms-on-mobile-devices-with-android-os/, Visited in Nov. 2018.

[36] Statista. https://www.statista.com/chart/7478/android-is-the-most-vulnerable-operating-system/, Visited in Nov. 2018.

[37] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro. Droidsieve: Fast and accurate classification of obfuscated android malware. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 309–320. ACM, 2017.

[38] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS*, 2015.

[39] R. Unuchek. Svpeng report. https://securelist.com/a-new-era-in-mobile-banking-trojans/79198/, Visited in Nov. 2018.

[40] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 590–604. IEEE, 2014.

[41] L. K. Yan and H. Yin. DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic android malware analysis. In *USENIX Security*, 2012.

[42] ZDNET. Svpeng report. https://www.zdnet.com/article/this-android-banking-malware-steals-data-by-exploiting-smartphone-accessibility-services/, Visited in Nov. 2018.

[43] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1105–1116. ACM, 2014.

[44] ZScaler. https://www.zscaler.com/blogs/research/new-android-marcher-variant-posing-adobe-flash-player-update, Visited in Nov. 2018.


## BIOGRAPHIES

**Chongyang Bai** is a second-year Ph.D. student at Dartmouth College advised by Prof. V.S. Subrahmanian. He obtained a BS in Computational Mathematics and BEng in Computer Science from University of Science and Technology of China in 2016. From 2015 to 2016, he was a research intern at Microsoft Research Asia. His research interests are machine learning and social network analysis.

**Qian Han** is a third-year Ph.D. student at Dartmouth College advised by Prof. V.S. Subrahmanian. He received a BEng in department of Electronic Engineering from Tsinghua University in 2016. During 2015, he spent 3 months as a visiting research assistant at Nanyang Technological University, Singapore. His research interests lie in cybersecurity, data-mining, game theory, and social network analysis.

**Ghita Mezzour** is an Assistant Professor at the International University of Rabat. She received a Ph.D. in Electrical and Computer Engineering from Carnegie Mellon University in 2015. She received a MS and BS in Communication Systems for Ecole Polytechnique Federale de Lausanne in 2008 and 2006 respectively. Her research interests include cybersecurity, big data, and social network analysis. Prof. Mezzour was selected as a Rising Star by MIT's Electrical Engineering and Computer Science Department in November 2015 and held a visiting position at the University of Maryland at College Park during Summer 2017.

**Fabio Pierazzi** is a Visiting Research Associate at King's College London and PostDoc at Royal Holloway, University of London, UK. He completed PhD in Computer Science at University of Modena and Reggio Emilia (Italy) in March 2017. During 2016, he spent 10 months as a visiting research scholar at University of Maryland, College Park, MD, USA. His research interests focus on machine learning for security, with focus on malware analysis and intrusion detection.

**V.S. Subrahmanian** is the Dartmouth College Distinguished Professor in Cybersecurity, Technology, and Society and Director of the Institute for Security, Technology, and Society at Dartmouth. He previously served as a Professor of Computer Science at the University of Maryland from 1989-2017 where he created and headed both the Lab for Computational Cultural Dynamics and the Center for Digital International Government. He also served for 6+ years as Director of the University of Maryland's Institute for Advanced Computer Studies. Prof. Subrahmanian is an expert on big data analytics including methods to analyze text/geospatial/relational/social network data, learn behavioral models from the data, forecast actions, and influence behaviors with applications to cybersecurity and counterterrorism. He has written five books, edited ten, and published over 300 refereed articles. He is a Fellow of the American Association for the Advancement of Science and the Association for the Advancement of Artificial Intelligence and received numerous other honors and awards. His work has been featured in numerous outlets such as the Baltimore Sun, the Economist, Science, Nature, the Washington Post, American Public Media. He serves on the editorial boards of numerous journals including Science, the Board of Directors of the Development Gateway Foundation (set up by the World Bank), SentiMetrix, Inc., and on the Research Advisory Board of Tata Consultancy Services. He previously served on DARPA's Executive Advisory Council on Advanced Logistics and as an ad-hoc member of the US Air Force Science Advisory Board.

# DBank: Predictive Behavioral Analysis of Recent Android Banking Trojans

Chongyang Bai, Qian Han, Ghita Mezzour, Fabio Pierazzi, and V.S. Subrahmanian

## Online Supplementary Material

## Appendix A
## Baseline Android Feature Sets

We hereby provide details of the lightweight static and dynamic features that we compare against and that have been used in past research for scalable malware detection [9], [4], [2], [8]. In particular, we consider three types of *static* features (extracted by analyzing an application metadata and code, without executing the application) and one type of *dynamic* features (obtained by running the application).

### A.1 Manifest

The *Manifest* feature set consists of matadata information associated with Android applications. These features are extracted from the Android `Manifest`, that is a file declaring the name of application, the requested permissions, and some auxiliary information about code declarations (e.g., name and type of Java classes implemented). In particular the Manifest feature set consists of: filesize, signature-hash of the application (to estimate the authorship), and number of Android components (Activities, Receivers, Intents, Content Providers, Services). In addition, the set contains information about requested Android permissions as binary vectors (where a permission is 1 if requested by the application, and 0 if not). The manifest features reliably replicate the features described in [4], which are a direct extension of other popular works (e.g., [2], [10]).

### A.2 API Class and API Package

In the Android system, API libraries to interact with the operating system and the hardware sensors (e.g., read and write operations) are organized into programming packages, such as `android.os` and `android.telephony`. Each package contains one or more classes, such as `android.telephony.CellLocation`. Past literature has often considered API classes and packages as feature sets (e.g., [8], [2], [10]).


- *C. Bai, Q.Han and V.S. Subrahmanian are with the Department of Computer Science and the Institute for Security, Technology, and Society, Dartmouth College, Hanover, NH 03755, USA. G. Mezzour is with the Dept. of Computer Science and Logistic and the TICLab, Universite Internationale de Rabat, Sala El Jadida, Morocco. F. Pierazzi is with King's College London and Royal Holloway, University of London, UK.*
- **Corresponding author:** *Professor V.S. Subrahmanian.*


In particular, here for each application we consider the total number of times (i.e., frequency) in which a method or class of a certain package as been called (as *API Package* feature set) and the number of times in which a method from a particular class has been called (as *API Class* feature set). As in prior work [9], [4], [8], we do not consider method call frequencies because it is too low-level, it increases too much the size of the feature space, and we aim to avoid overfitting of the training data characteristics.

An important observation is that we do not consider advanced API call graphs such as control-flow graphs in FlowDroid [3] or dependency graphs like in DroidSIFT [12], because they are very computationally intensive to extract and hence not scalable. For example, FlowDroid can take up to an hour to extract features from an individual application. Instead, we focus on API class frequency that have been successfully adopted as a basis for building features, and are easy to obtain in a scalable way.

### A.3 Dynamic

The *Dynamic* features are extracted by executing the applications on Koodous [1], an online service for Android malware sandboxing and analysis. In particular, Koodous executes the Android application for 60 seconds, and record the major activities by monitoring the app behavior with `cuckoo` and `Droidbox` sandboxes. As in prior work [4], [9], we consider 1-gram, 2-gram and 3-gram of simplified system call activities associated with the following events: Read/Write operations, System operations (e.g., class loading, start of background service), Network operations, and SMS activity. For each app, the n-gram frequency counts are considered as part of the feature vector. We replicate the dynamic features described in more detail in [4], which are inspired by [9].

An important observation is that we do not consider dynamic coverage or advanced application stimulation because it is an open challenge [11] outside the scope of this paper. Instead, we focus on lightweight dynamic analysis that can easily be performed at scale.

## Appendix B
## Listing and Description of Most Important Features

We report two tables that summarize the most important features in separating ABTs from Goodware (Table 1) and

in separating ABTs from other malware (Table 2). We then report the description of the 10 most important features characterizing `FakeToken` in Table 3, in terms of features distinguishing FakeToken from goodware and from other malware. NOTE: **Descriptions with gray background are reported *verbatim* from the official Android documentation [5], [6], [7]. We report them here just as a quick reference for the reader.**

## APPENDIX C
## RESULTS WITH ISOMORPHIC SAMPLES

Table 4 and Table 5 show the prediction AUC (%) and FPR (%) with isomorphic samples. We report only individual features because it already shows that the performance are highly inflated (unlike the ones we reported in Section 4, which did not include isomorphic samples).

## APPENDIX D
## FEATURE HISTOGRAMS

We report the full histograms reporting the key features of ABTs vs. other categories. In particular, these histograms refer to the features with highest *importance scores* accorting to DT-based classifiers, as they are the best performing ones in our dataset. Figure 1 reports the top-25 feature histograms of ABT vs goodware. Figure 2 reports the top-25 feature histograms of ABT vs other malware. Figure 3 reports top-10 features of `FakeToken` vs goodware and the top-10 features of `FakeToken` vs other-malware. Similarly, Figure 4 for `Svpeng`, Figure 5 for `Asacub`, Figure 6 for `BankBot`, and Figure 7 for `Marcher`.

## APPENDIX E
## ADVERSARY FEATURE DISTANCES

We report additional plots for the feature distances in the robustness experiment in Section 5 of the main text. In particular, Figure 8 reports the results with respect to the Manhattan distance, Figure 9 reports the results with respect to the Cosine distance, and Figure 10 reports the results with respect to the Chebyshev distance.

## REFERENCES


[1] Koodous. https://koodous.com/. Accessed: November 2018.

[2] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *NDSS*, 2014.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm SIGPLAN Notices*, 2014.

[4] T. Chakraborty, F. Pierazzi, and V. S. Subrahmanian. Ec2: Ensemble clustering and classification for predicting android malware families. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[5] A. P. Documentation. Api package list. https://developer.android.com/reference/android/Manifest.permission, Visited in Mar. 2019.

[6] A. P. Documentation. Api package list. https://developer.android.com/reference, Visited in Mar. 2019.

[7] A. P. Documentation. Api package list. https://developer.android.com/reference/packages, Visited in Mar. 2019.

[8] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini. MaMaDroid: Detecting android malware by building markov chains of behavioral models. *NDSS*, 2017.

[9] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In *DIMVA*. Springer, 2008.

[10] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro. Droidsieve: Fast and accurate classification of obfuscated android malware. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 309–320. ACM, 2017.

[11] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS*, 2015.

[12] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1105–1116. ACM, 2014.

TABLE 1: Description of Top-25 features for ABT vs. Goodware. NOTE: **Descriptions with gray background have been reported *verbatim* from the official Android documentation [5], [6], [7].**

| Feature | Description |
|---|---|
| permission: RECEIVE_SMS | Allows an application to receive SMS messages. |
| android.view | Provides classes that expose basic user interface classes that handle screen layout and interaction with the user. |
| filesize | The filesize of the application. |
| android.widget | The widget package contains (mostly visual) UI elements to use on Application screen. |
| num_std_permissions | The number of standard permissions (manifest permissions in Android developers documentation) used in application. |
| permission: READ_PHONE_STATE | Allows read only access to phone state, including the phone number of the device, current cellular network information, the status of any ongoing calls, and a list of any PhoneAccounts registered on the device. |
| android.net | Classes that help with network access, beyond the normal java.net.* APIs. |
| android.text | Provides classes used to render or track text and text spans on the screen. |
| javax.net | Provides classes for networking applications. These classes include factories for creating sockets. |
| java.text | Provides classes and interfaces for handling text, dates, numbers, and messages in a manner independent of natural languages. |
| num_non_std_permissions | The number of non-standard permissions (not available in Android developers documentation) used in application. |
| permission: SYSTEM_ALERT_WINDOW | Allows an app to create windows using the type WindowManager.LayoutParams. TYPE_APPLICATION_OVERLAY, shown on top of all other apps. |
| android.app.admin | Provides device administration features at the system level, allowing you to create security-aware applications that are useful in enterprise settings, in which IT professionals require rich control over employee devices. |
| permission: (UN)MOUNT_FILESYSTEMS | Allows mounting and unmounting file systems for removable storage. |
| num_receivers | The number of receivers (Broadcast receivers enable applications to receive intents that are broadcast by the system or by other applications, even when other components of the application are not running. |
| android.content | Contains classes for accessing and publishing data on a device. |
| android.view.inputmethod | Framework classes for interaction between views and input methods (such as soft keyboards). |
| android.media.session | Allows interaction with media controllers, volume keys, media buttons, and transport controls. |
| android.appwidget | Contains the components necessary to create "app widgets", which users can embed in other applications (such as the home screen) to quickly access application data and services without launching a new activity. |
| java.lang.reflect | Provides classes and interfaces for obtaining reflective information about classes and objects. Reflection allows programmatic access to information about the fields, methods and constructors of loaded classes, and the use of reflected fields, methods, and constructors to operate on their underlying counterparts, within security restrictions. |
| android.telephony | Provides APIs for monitoring the basic phone information, such as the network type and connection state, plus utilities for manipulating phone number strings. |
| write b'appsflyer-data.xml' | Dynamic feature: writing binary file b'appsflyer-data.xml during operation |
| android.content.res | Contains classes for accessing application resources, such as raw asset files, colors, drawables, media, or other files in the package, plus important device configuration details (orientation, input types, etc.) that affect how the application may behave. |
| servicestart b'PG_Service' | Dynamic feature: start service PG_Service during operation |
| read b'com.q3600.app.outsourcing.binding-1.apk' | Dynamic feature: read file com.q3600.app.outsourcing.binding-1.apk during operation |

TABLE 2: Description of Top-25 features for ABT vs. Other-Malware. NOTE: **Descriptions with gray background have been reported *verbatim* from the official Android documentation [5], [6], [7].**

| Feature | Description |
|---|---|
| permission: READ_SMS | Allows an application to read SMS messages. |
| android.app.admin | Provides device administration features at the system level, allowing you to create security-aware applications that are useful in enterprise settings, in which IT professionals require rich control over employee devices. |
| filesize | The filesize of the application |
| dalvik.system | Provides utility and system information classes specific to the Dalvik VM |
| permission: GET_TASKS | Allows an application to get information about the currently or recently running tasks. |
| java.io | Provides for system input and output through data streams, serialization and the file system. |
| android.view | Provides classes that expose basic user interface classes that handle screen layout and interaction with the user. |
| num_non_std_permissions | The number of non-standard permissions (not available in Android developers documentation [6]) used in application. |
| android.media | Provides classes that manage various media interfaces in audio and video. |
| android.app | Contains high-level classes encapsulating the overall Android application model. |
| num_intents | The number of intents (An Intent is a simple message object that is used to communicate between android components such as activities, content providers, broadcast receivers and services) in application. |
| android.database | Contains classes to explore data returned through a content provider. |
| android.view.animation | Provides classes that handle tweened animations. |
| android.net | Classes that help with network access, beyond the normal java.net.* APIs. |
| android.graphics | Provides low level graphics tools such as canvases, color filters, points, and rectangles that let you handle drawing to the screen directly. |
| android.content | Contains classes for accessing and publishing data on a device. |
| num_services | The number of services (A Service is an application component that can perform long-running operations in the background, and it doesn't provide a user interface) in application |
| android.os | Provides basic operating system services, message passing, and inter-process communication on the device. |
| android.hardware | Provides support for hardware features, such as the camera and other sensors. |
| java.util.zip | Provides classes for reading and writing the standard ZIP and GZIP file formats. |
| num_receivers | The number of receivers (Broadcast receivers enable applications to receive intents that are broadcast by the system or by other applications, even when other components of the application are not running. ) in application. |
| permission: CALL_PHONE | Allows an application to initiate a phone call without going through the Dialer user interface for the user to confirm the call. |
| android.preference | Provides classes that manage application preferences and implement the preferences UI. |
| android.telephony | Provides APIs for monitoring the basic phone information, such as the network type and connection state, plus utilities for manipulating phone number strings. |
| java.util | Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array). |

TABLE 3: Description of top-10 features for `FakeToken` vs. Goodware (left table) and `FakeToken` vs. Other-malware (right table). NOTE: **Descriptions with gray background have been reported *verbatim* from the official Android documentation [5], [6], [7].**

| Feature | Description |
|---|---|
| android.content | Contains classes for accessing and publishing data on a device. |
| permission: WRITE_SMS | Allows the app to write to SMS messages stored on your phone or SIM card. Malicious apps may delete your messages. |
| filesize | The filesize of the application. |
| android.text | Provides classes used to render or track text and text spans on the screen. |
| num_std_permissions | The number of standard permissions (manifest permissions in Android developers documentation) used in application. |
| read b'/data/' | Dynamic feature: read file com.q3600.app.outsourcing.b inding-1.apk during operation |
| permission: GET_TASKS | Allows an application to get information about the currently or recently running tasks. |
| dexclass b'com.example.testcallchange-1.apk' | Dynamic feature: read dex file (Compiled Android application code file) com.example.testcallchange-1.apk during operation. |
| permission: READ_PHONE_STATE | Allows read only access to phone state, including the phone number of the device, current cellular network information, the status of any ongoing calls, and a list of any PhoneAccounts registered on the device. |
| read b'com.jeujlu.gnvldarosz-1.apk' | Dynamic feature: read file com.jeujlu.gnvldarosz-1.apk during operation. |

| Feature | Description |
|---|---|
| num_intents | The number of intents (An Intent is a simple message object that is used to communicate between android components such as activities, content providers, broadcast receivers and services) in application. |
| java.io | Provides for system input and output through data streams, serialization and the file system. |
| num_activities | The number of activities (The Activity class is a crucial component of an Android app, and the way activities are launched and put together is a fundamental part of the platform's application model.) in application. |
| java.util | Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array). |
| java.lang | Provides classes that are fundamental to the design of the Java programming language. |
| num_std_permissions | The number of standard permissions (manifest permissions in Android developers documentation) used in application. |
| android.app.admin | Provides device administration features at the system level, allowing you to create security-aware applications that are useful in enterprise settings, in which IT professionals require rich control over employee devices. |
| android.content | Contains classes for accessing and publishing data on a device. |
| android.view | Provides classes that expose basic user interface classes that handle screen layout and interaction with the user. |
| android.os | Provides basic operating system services, message passing, and inter-process communication on the device. |

TABLE 4: AUC (%) and FPR (%) on ABT vs. Goodware with isomorphic samples. Cells with gray background highlight the best AUC in the group. We report results for the following features: TSG, Manifest (M), Dynamic (D), API package (AP), and API class (AC).

| | | KNN | | LR | | DT | | NB | | RF | | GBDT | | MLP | | SVM | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Features** | AUC | FPR | AUC | FPR | AUC | FPR | AUC | FPR | AUC | FPR | AUC | FPR | AUC | FPR | AUC | FPR |
| INDIVIDUAL | M (Manifest) | 96.4 | 9.0 | 99.4 | 1.7 | 97.8 | 3.2 | 96.8 | 4.4 | 99.8 | 0.5 | 98.7 | 2.7 | 50.0 | 40.0 | 44.0 | 60.5 |
| | AP (API Package) | 98.6 | 6.1 | 98.6 | 6.3 | 98.1 | 2.9 | 95.8 | 10.5 | **99.9** | 0.9 | 99.0 | 2.7 | 94.2 | 7.1 | 88.3 | 15.2 |
| | AC (API Class) | 98.5 | 6.1 | 99.1 | 2.7 | 98.5 | 2.1 | 94.5 | 9.6 | **99.9** | 0.9 | 99.1 | 1.9 | 94.7 | 11.8 | 96.5 | 6.5 |
| | D (Dynamic) | 83.7 | 36.9 | 96.0 | 0.9 | 94.6 | 8.3 | 94.9 | 4.7 | 97.1 | 4.6 | 96.8 | 10.4 | 96.5 | 14.3 | 96.1 | 14.7 |
| | TSG (Ours) | 98.7 | 6.4 | 98.9 | 3.5 | 98.1 | 2.8 | 93.5 | 10.0 | **99.9** | 0.7 | 99.1 | 2.6 | 64.8 | 78.0 | 76.7 | 29.3 |

TABLE 5: AUC (%) and FPR (%) on ABT vs. Other-malware with isomorphic samples. Cells with gray background highlight the best AUC in the group. We report results for the following features: TSG, Manifest (M), Dynamic (D), API package (AP), and API class (AC).

| | | KNN | | LR | | DT | | NB | | RF | | GBDT | | MLP | | SVM | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Features** | AUC | FPR | AUC | FPR | AUC | FPR | AUC | FPR | AUC | FPR | AUC | FPR | AUC | FPR | AUC | FPR |
| INDIVIDUAL | M (Manifest) | 87.6 | 26.1 | 96.0 | 13.7 | 95.4 | 6.0 | 88.6 | 14.2 | 99.0 | 4.6 | 98.5 | 6.8 | 50.0 | 50.0 | 49.0 | 57.7 |
| | AP (API Package) | 97.9 | 7.8 | 93.8 | 11.9 | 97.0 | 7.1 | 85.9 | 20.2 | **99.3** | 5.9 | 98.6 | 7.7 | 92.9 | 24.5 | 65.1 | 18.0 |
| | AC (API Class) | 97.7 | 8.7 | 97.0 | 12.8 | 96.9 | 10.4 | 51.6 | 81.6 | **99.3** | 5.3 | 98.6 | 6.7 | 89.7 | 40.9 | 98.1 | 6.4 |
| | D (Dynamic) | 88.4 | 5.8 | 95.1 | 24.8 | 92.0 | 19.8 | 93.7 | 5.3 | 96.6 | 22.0 | 95.5 | 22.6 | 95.4 | 20.5 | 95.3 | 24.1 |
| | TSG (Ours) | 97.6 | 8.6 | 97.0 | 8.7 | 96.6 | 7.6 | 80.0 | 25.4 | **99.3** | 6.1 | 98.5 | 7.5 | 73.7 | 65.9 | 74.9 | 20.4 |

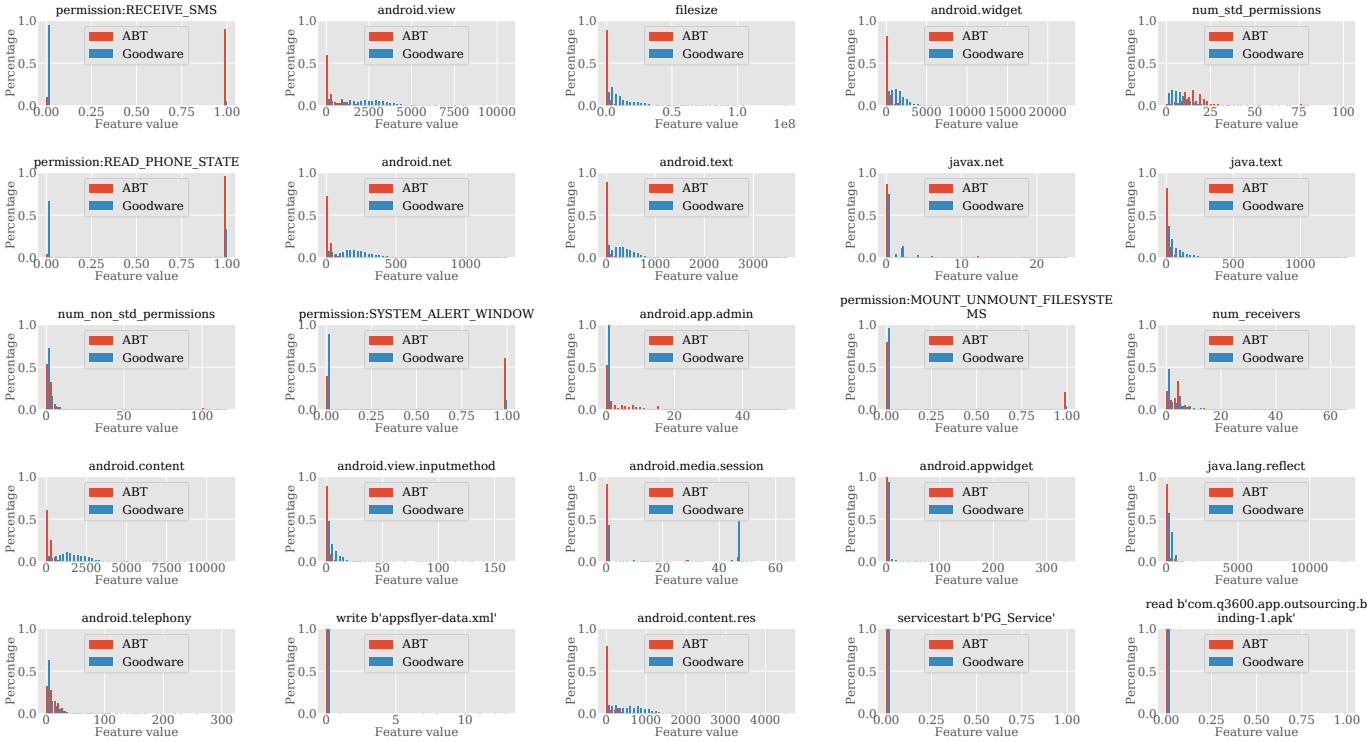



Fig. 1: Top-25 features of ABT vs. Goodware.

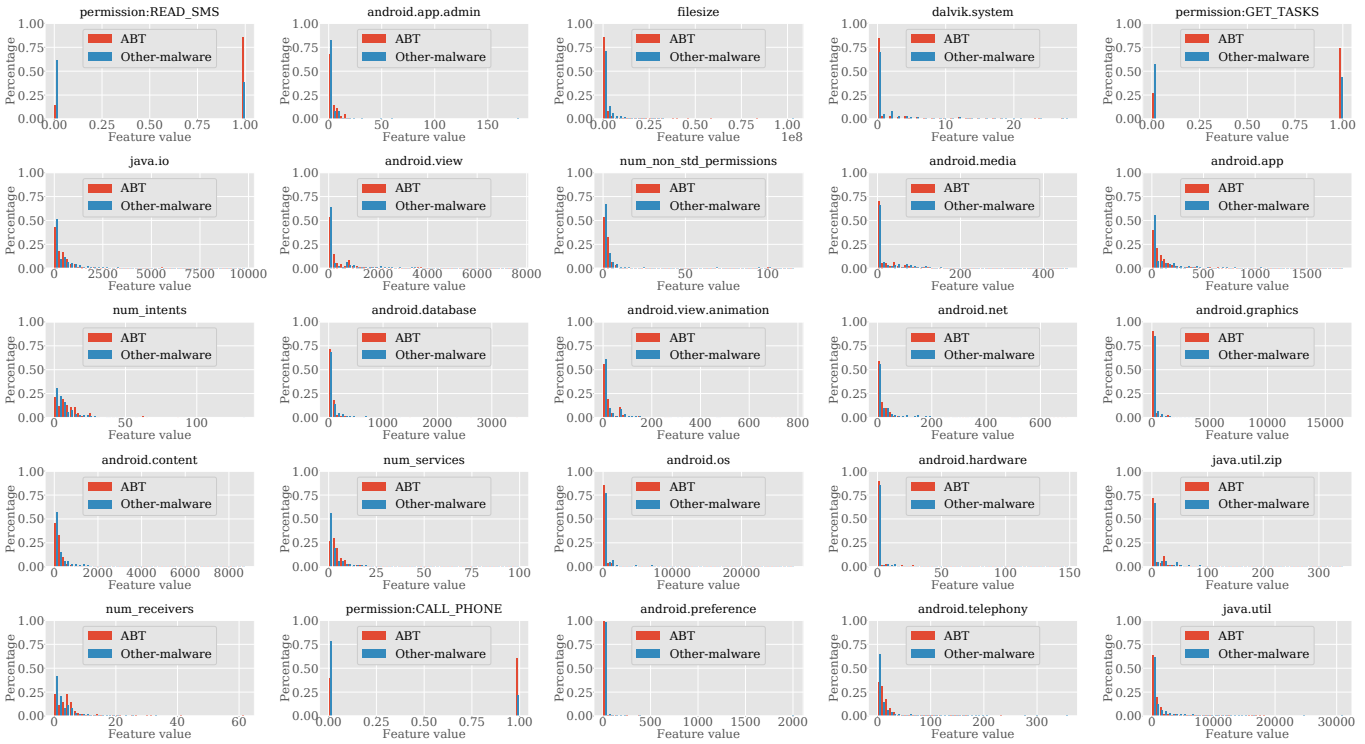
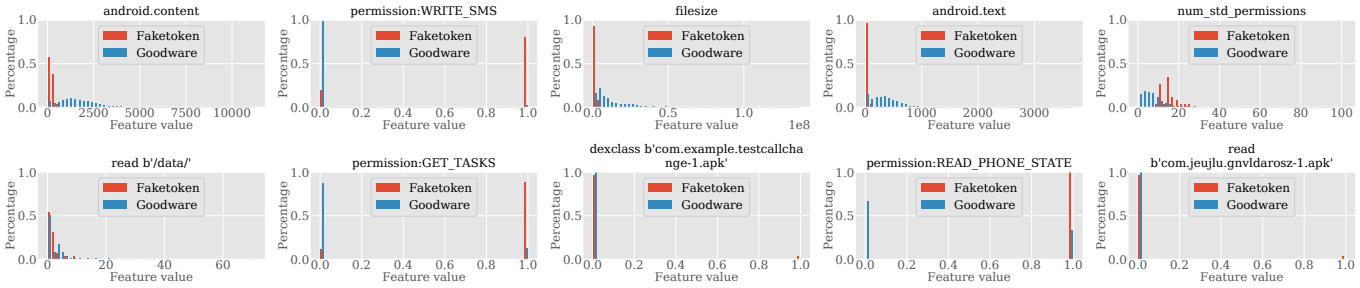
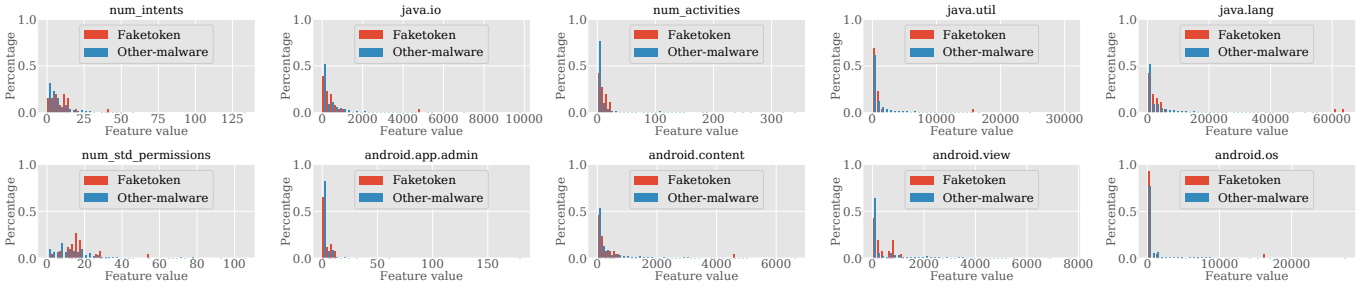



Fig. 2: Top-25 features of ABT vs. Other-Malware.

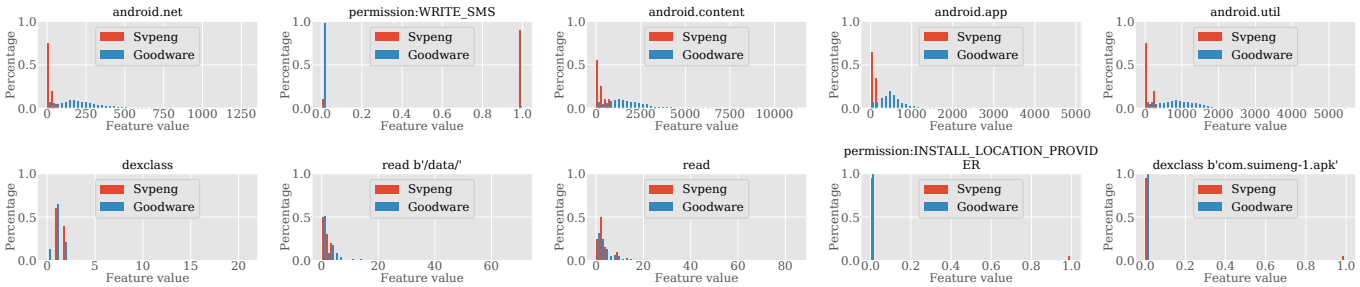



(a) `FakeToken` vs. Goodware

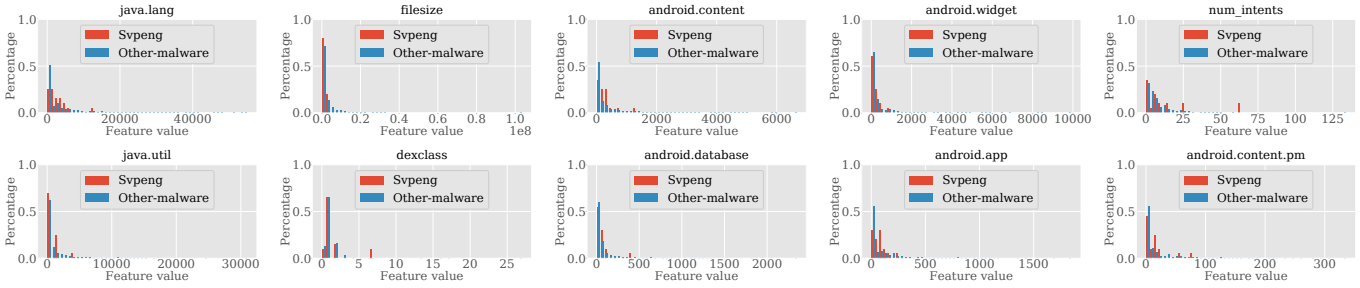



(b) `FakeToken` vs. Other-malware

Fig. 3: Top-10 features of `FakeToken`.

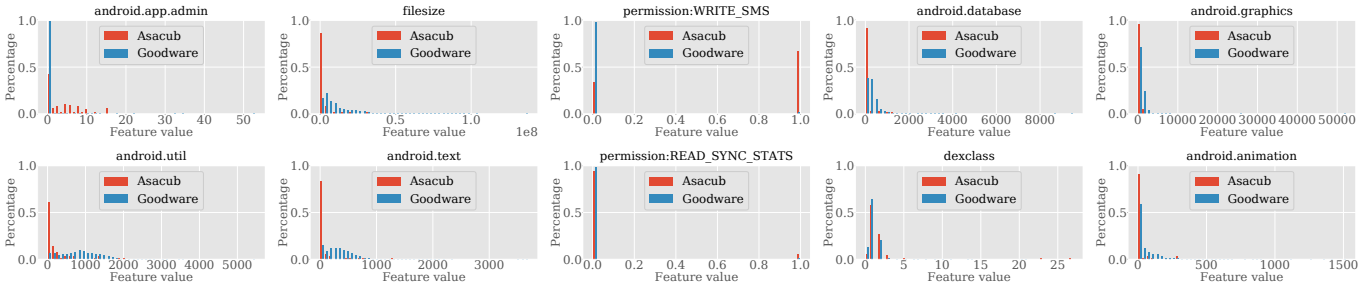



(a) `Svpeng` vs. Goodware

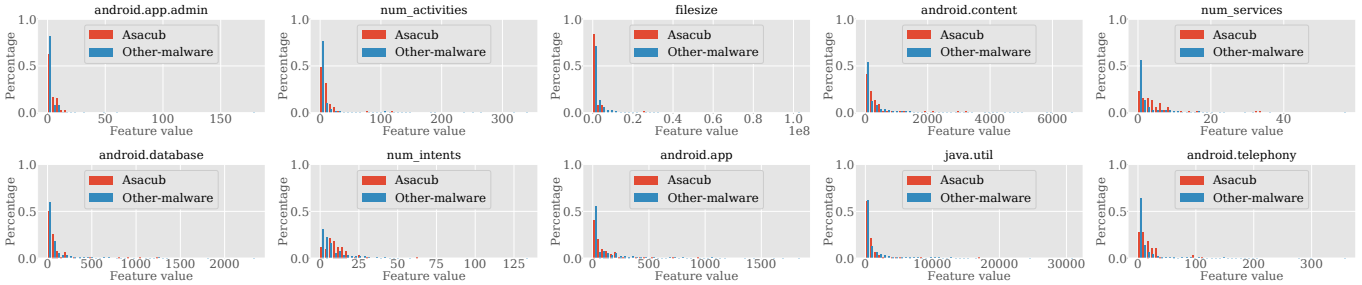



(b) `Svpeng` vs. Other-malware

Fig. 4: Top-10 features of `Svpeng`.

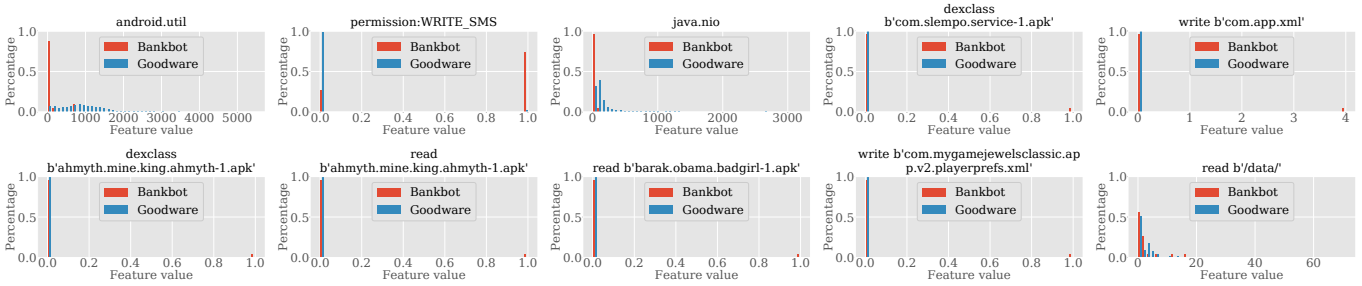



(a) `Asacub` vs. Goodware

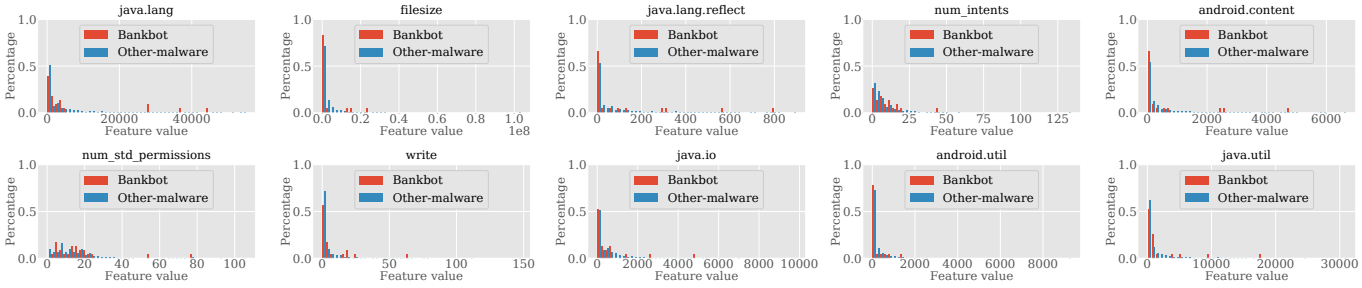



(b) `Asacub` vs. Other-malware

Fig. 5: Top-10 features of `Asacub`.

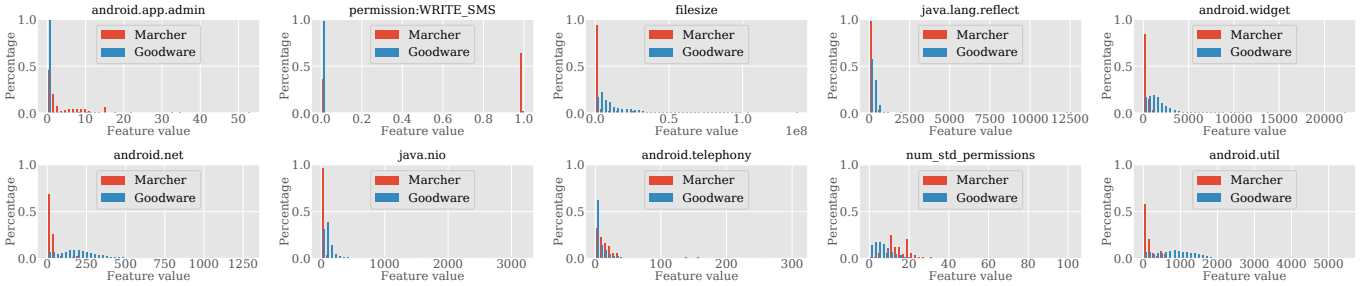



(a) `BankBot` vs. Goodware

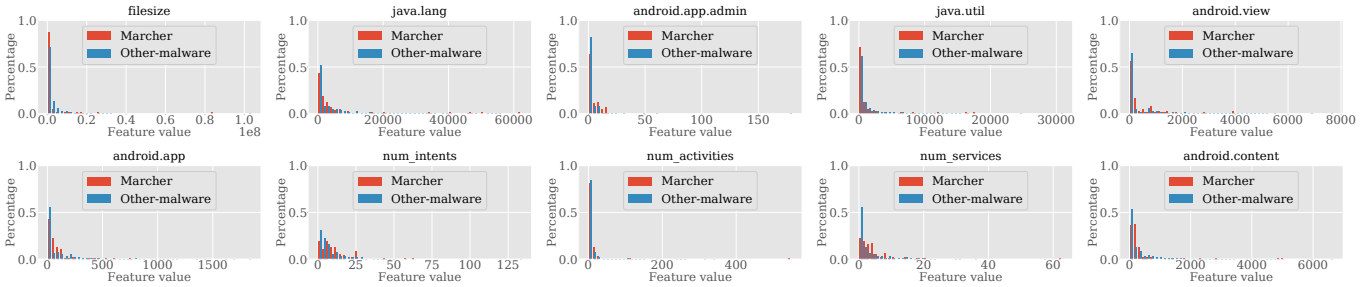



(b) `BankBot` vs. Other-malware

Fig. 6: Top-10 features of `BankBot`.

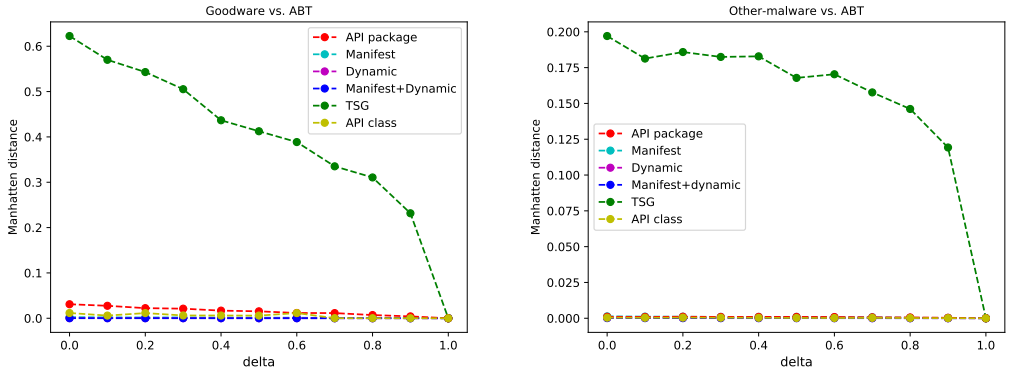



(a) `Marcher` vs. Goodware



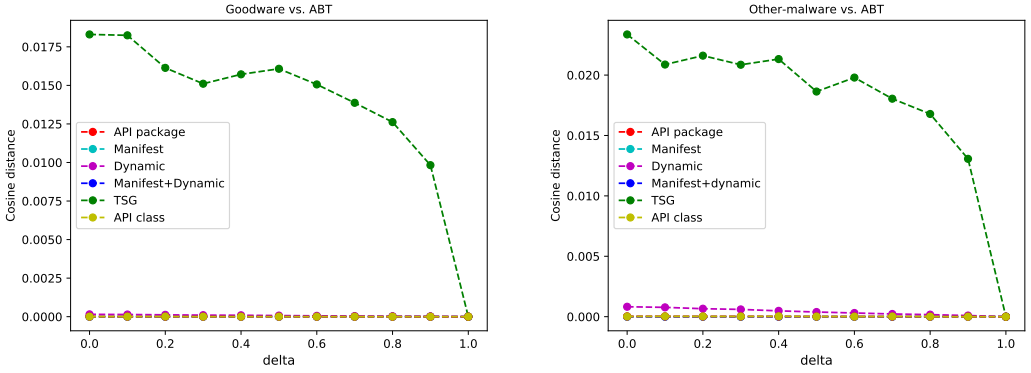

(b) `Marcher` vs. Other-malware

Fig. 7: Top-10 features of `Marcher`.



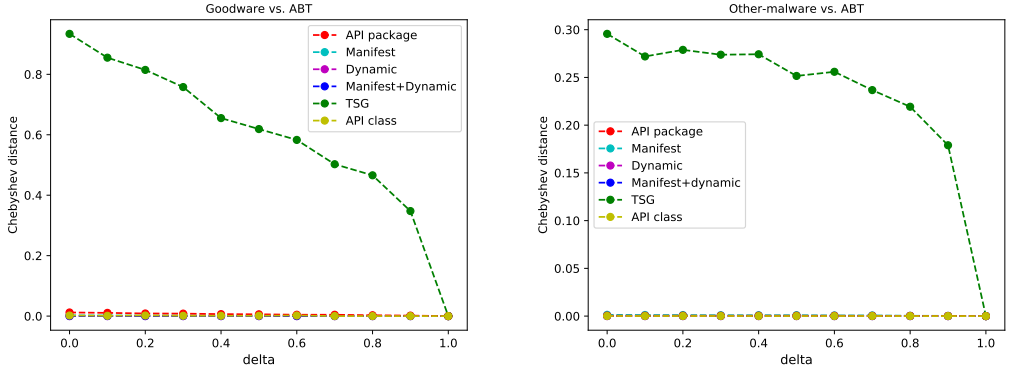

(a) Goodware vs. ABT (b) Other-malware vs. ABT

Fig. 8: Manhattan distance among attacker's and defender's feature sets centroids. High values imply higher robustness.




(a) Goodware vs. ABT (b) Other-malware vs. ABT

Fig. 9: Cosine distance among attacker's and defender's feature sets centroids. High values imply higher robustness.




(a) Goodware vs. ABT (b) Other-malware vs. ABT

Fig. 10: Chebyshev distance among attacker's and defender's feature sets centroids. High values imply higher robustness.